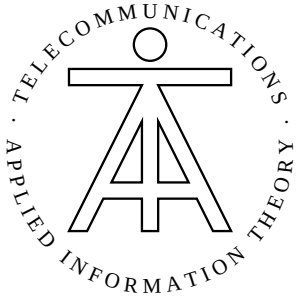
Design und Implementierung eines Frameworks für Simulationen in der Kanalcodierung mit Anbindung an ein verteiltes System

Diplomarbeit

von

Alexander Bernauer



Abteilung Telekommunikationstechnik
und Angewandte Informationstheorie
Universität Ulm
April 2006

D/2005/XL/2

Abteilung Telekommunikationstechnik
und Angewandte Informationstheorie
Universität Ulm

DIPLOMARBEIT

Design und Implementierung eines Frameworks für Simulationen in der Kanalcodierung mit Anbindung an ein verteiltes System

Erläuterungen:

Um in der Kanalcodierung theoretische Ergebnisse zu verifizieren werden Simulationen für die Berechnung der Restbit- und -blockfehlerraten verwendet. Dabei werden entweder graphische Simulationstools wie „ML Designer“ verwendet oder es werden komplette Simulationsumgebungen in einer Programmiersprache wie C/C++ oder „Matlab“ erstellt. Beide Möglichkeiten haben ihre Vor- und Nachteile.
Bestehende Simulationstools sind meist recht umfangreich und dementsprechend unübersichtlich. Die Einarbeitung nimmt daher oftmals viel Zeit in Anspruch. Desweiteren handelt es sich bei diesen Tools häufig um kommerzielle Software, wodurch hohe Lizenzkosten anfallen.
Auf der anderen Seite sind vom Anwender selbst geschriebene Simulationsumgebungen zwar frei, haben aber den Nachteil, dass von anderen Benutzern sehr viel Progammierkenntnis verlangt wird, um Modifikationen vornehmen zu können. Die komfortable graphische Benutzeroberfläche kommerzieller Tools entfällt in diesem Fall.
Im Rahmen dieser Diplomarbeit soll daher ein Framework für Simulationen in der Kanalcodierung implementiert werden. Dabei soll die Anbindung an ein verteiltes System gegeben sein, um Simulationen zu verwalten und in einem heterogenen Netzwerk zu verteilen, was jedoch Teil einer weiteren Diplomarbeit ist. Damit soll es Anwendern ohne tiefergehende Programmierkenntnisse möglich sein, auf einfache Weise Simulationen durchzuführen.
Zu diesem Zweck soll das Framework nur eine Beschreibung der Simulation mit beteiligten Modulen und Parametern entgegennehmen und die Simulation dann eigenständig durchführen. Zudem soll für die Entwicklung neuer Module nur sehr wenig Wissen über die Interna des Frameworks notwendig sein. Die Einbindung neuer Module ist damit für programmiererfahrenere Anwender relativ einfach zu bewerkstelligen.

Abgabetermin: 17. April 2006

Bearbeiter: Alexander Bernauer

Betreuung: Prof. Dr.-Ing. M. Bossert
Prof. Dr.-Ing. F. J. Hauck

Dipl.-Ing. A. Hof
Dipl.-Inform. S. Schober

Katalognr.: D/2005/XL/2

Erklärung

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe angefertigt habe.

Ulm, den 17. Juli 2006

Alexander Bernauer

Danksagung

Mein größter Dank geht an meine Eltern, die mir durch ihre jahrelange finanzielle Unterstützung mein Studium ermöglicht haben. Außerdem möchte ich allen Mitwirkenden der Universität Ulm für ihren Hilfe bei dieser Diplomarbeit danken. Diese sind die Gutachter Prof. Dr.-Ing. Martin Bossert und Prof. Dr.-Ing. Franz J. Hauck und meine Betreuer Dipl.-Ing. Axel Hof und Dipl.-Inform. Steffen Schober.

Diese Diplomarbeit wurde von einigen Diskussionen mit Aussenstehenden begleitet, was für viele Entscheidungen ausschlaggebend oder hilfreich war. Die wesentlichen Personen waren Volker Birk, Andreas Pretzsch, Michael Feiri und die Kommilitonen aus dem Linux-Pool. Ich bin diesen Leuten für ihre Zeit und ihre Gedanken dankbar.

Desweiteren danke ich allen Korrekturlesern für ihre Arbeit. Zuerst ist hier mein Bruder Andreas Bernauer zu nennen, der durch frühzeitige und kritische Korrekturen den Aufbau dieser Arbeit erheblich beinflusst hat. Andere Korrekturleser waren Liesa Keizer, Markus Schaber und Ulrich Dangel. Besonders danken möchte ich Anne Keizer und Nico Brinker für die abschließende Korrektur von Rechtschreibfehlern am Osterwochenende.

Für die Entwicklung und die Implementierung der in dieser Arbeit vorgestellten Software wurde sehr viel freie Software von Dritten eingesetzt. Deshalb möchte ich mich an dieser Stelle bei den Entwicklern und Maintainern für ihre jeweilige Arbeit bedanken.

Schließlich geht noch ein besonderer Dank an Liesa Keizer, die mich vorallem gegen Ende hin, trotz zahlreicher Entbehrungen ständig unterstützt hat.

Inhaltsverzeichnis

Inhaltsverzeichnis **VII**

1 Einführung **1**
1.1 Aufteilung und Gliederung 2
1.2 Die Anforderungen 2

2 Grundlagen und Stand der Technik **5**
2.1 Simulationsprogramme 5
2.2 Simulationsalgorithmen 6
2.3 Multitasking 8
2.4 Hypergraphen 9
2.5 C++ 10
2.6 Numerische Bibliotheken 11

3 Systementwurf **13**
3.1 Die Simulation 13
3.1.1 Das Fallbeispiel 14
3.1.2 Das Modell 14
3.1.3 Das Scheduling 16
3.1.4 Das Ende einer Simulation 18
3.1.5 Verteilung 20
3.1.6 Feedback-Kanten 21
3.1.7 Phasen 22
3.1.8 Mehrere Schreiber 24
3.1.9 Zwischenzustände 25
3.1.10 Der Lebenszyklus eines Knotens 26
3.1.11 Die Simulationsbeschreibung 27
3.2 Das verteilte System 28
3.2.1 Manager 28
3.2.2 Simulationsobjekte 28
3.2.3 Der Generator 31
3.2.4 Source-Service 31
3.2.5 Proxies 32
3.2.6 Der Breeze-Server 33

4 Implementierung **35**
4.1 Die Wahl der Programmiersprache . . . 35
4.2 Streams . . . 36
4.2.1 Der Puffer . . . 37
4.2.2 Die Größe eines Puffers . . . 38
4.2.3 Basistypen . . . 38
4.2.4 Arbeitstypen . . . 39
4.2.5 Vermeidung von Konvertierungen . . . 41
4.3 Ports . . . 42
4.3.1 Die Vector-Klassen . . . 42
4.3.2 Die Adapter-Klasse . . . 44
4.4 Module . . . 46
4.4.1 Verwaltungsklassen . . . 46
4.4.2 Die Entwickler API . . . 47
4.5 Das Scheduling . . . 54
4.5.1 Vernetzung . . . 55
4.5.2 Ereignisse . . . 56
4.5.3 Die call-Funktion . . . 56

5 Messungen **59**
5.1 Testkriterien . . . 59
5.2 Wiederholung der Messungen . . . 60
5.3 Verwaltungsaufwand . . . 60
5.4 Konvertierungen . . . 61
5.4.1 Erhöhung der Konvertierungshäufigkeit . . . 62
5.4.2 Verwendung der Optimierung . . . 62
5.5 Erhöhung der Blockgröße . . . 63
5.5.1 Deaktivierung der Optimierung . . . 65
5.6 Erhöhung der Datenmenge . . . 66

6 Schlußfolgerungen und Ausblicke **69**
6.1 Erfüllung der Anforderungen . . . 69
6.2 Anwendungsgebiete . . . 69
6.3 Lizenzrecht . . . 70
6.4 Ausblicke . . . 70
6.4.1 IceStorm . . . 71
6.4.2 Fehlersuche . . . 71
6.4.3 Unterstützung von manueller Parallelisierung . . . 71
6.4.4 Unterstützung für Matlab . . . 72
6.4.5 Beschleunigung des Generators . . . 72

7 Zusammenfassung **75**

A GNU PTH **77**
A.1 Zeitmessung der GNU PTH 77
A.1.1 Verwendung der GNU PTH 77
A.1.2 Verwendung eines Verteilers 78
A.1.3 Umgebung 79
A.1.4 Ergebnisse 79
A.2 Halbieren der Anzahl der Kontextwechsel 80
A.2.1 pth-fast.h 80
A.2.2 pth-fast.c 80
A.2.3 Eine beispielhafte main.c 83

B Dienste **85**
B.1 Generator 85
B.2 Breeze Server 86
B.2.1 Nachrichten 86
B.2.2 Konfiguration 87
B.2.3 Logdateien 87
B.2.4 publish 87
B.2.5 lastlog 88

C Verwendete Software **91**

Index **92**

Abbildungsverzeichnis **93**

Literaturverzeichnis **95**

1 Einführung

Die Telekommunikation spielt in der modernen Gesellschaft eine zentrale Rolle. Egal ob bei Steuerchips in Autos, Schiffen oder Flugzeugen, bei Signalprozessoren in Mobiltelefonen oder bei Controllern von Bussystemen in Computern oder Industrieanlagen, überall werden Informationen ausgetauscht und verarbeitet. Dabei steigen die Anforderungen an Geschwindigkeit und Robustheit unter immer kritischeren Bedingungen kontinuierlich an.

Forscher auf dem Gebiet der Informationstheorie sorgen seit Jahrzehnten immer wieder für Lösungen, die den wachsenden Ansprüchen gerecht werden. Einer der zentralen Schauplätze auf diesem Gebiet ist dabei die Kanalcodierung. Sie beschäftigt sich mit der Frage, wie man den Informationsaustausch durch kontrolliertes Hinzufügen von möglichst wenig Redundanz robust gegen Übertragungsfehler machen kann.

Die Komplexität der Systeme, die sich bei der Erforschung geeigneter Algorithmen ergeben, ist ohne Hilfsmittel allerdings kaum in den Griff zu bekommen. Eines dieser Hilfsmittel ist die Simulation. Dabei berechnet ein Computer anhand eines Modells möglichst exakt, wie sich das System bei einem echten Einsatz verhalten würde.

Es existiert also ein Markt für Simulationsprogramme und er wird reichlich bedient. Doch keine der Lösungen schöpft die Möglichkeiten, die sich durch verteiltes Rechnen ergeben, ausreichend aus, obwohl in den meisten Forschungseinrichtungen viele Computer vorhanden sind. Deshalb bleibt es dem Anwender überlassen, die Probleme zu lösen, die sich beim Einsatz in einem Netzwerk ergeben: es gibt keine automatisierte Verteilung, keine Ausfallsicherheit und keine zentrale Verwaltung für Ressourcen, Simulationen und Benutzer.

Diese Arbeit behandelt den Entwurf und die Implementierung von Lethe [1], einer Lösung dieser Probleme. Lethe ist zum einen ein verteiltes System mit einem zentralen Anwenderprogramm für die Erzeugung und Verwaltung von Simulationen. Neben der automatisierten Verteilung ermöglicht Lethe es, Simulationen im laufenden Betrieb auf andere Computer zu migrieren. Damit ist eine flexible Nutzung der zur Verfügung stehenden Ressourcen möglich. Außerdem erstellt Lethe in regelmäßigen Abständen Sicherungspunkte von Simulationen, die jederzeit auf einem beliebigen Computer fortgesetzt werden können. Lethe kümmert sich damit um die transparente Behandlung von Systemausfällen. Alle automatisierten Vorgänge können auch manuell gesteuert werden. Dabei sind Zugriffe auf das verteilte System durch ein Rechtesystem für Benutzer geschützt.

Zum anderen ist Lethe ein Framework für die Simulationen, das aus einer Programmierschnittstelle und eine Laufzeitumgebung besteht. Das wesentliche Kriterium für die Gestalt dieses Frameworks ist die Ermöglichung der im vorigen Absatz beschriebenen

[1] Der Name Lethe stammt aus der griechischen Mythologie. Er bezeichnet dort den Fluß des Vergessens im Reich von Hades, dem Gott der Unterwelt [20].

Fähigkeiten von Lethe. Dabei wird vor allem die Tatsache ausgenutzt, dass Lethe auf Anwendungen in der Kanalcodierung spezialisiert ist. Ein anderes wichtiges Kriterium ist es, die Wiederverwendbarkeit von Implementierungen zu maximieren. Deshalb existiert unter anderem ein Konzept für parametrisierbare Module.

Lethe kennt also zwei getrennte Rollen von Anwendern. Der Entwickler ist derjenige, der einen neuen Algorithmus erforschen möchte und ihn deshalb unter Verwendung des Frameworks implementiert. Der Benutzer ist dagegen derjenige, der auf eine gegebene Menge von implementierten Algorithmen zurückgreift und daraus mit Hilfe des Anwenderprogramms Simulationen erstellt, die von Lethe ausgeführt werden sollen.

Eines der Ziele von Lethe ist es, einen möglichst großen Anwenderkreis zu erreichen. Deshalb stellt es an das Netzwerk, in dem es eingesetzt wird, fast keine Anforderungen. Es benötigt lediglich POSIX-kompatible (ISO 9945) Systeme, wobei die einzelnen Systeme im Netzwerk verschieden sein dürfen. Außerdem steht Lethe unter der "GNU General Public Licence" [16] der "Free Software Foundation" [12], so dass so gut wie keine lizenzrechtlichen Hürden existieren.

1.1 Aufteilung und Gliederung

Das Lethe-Projekt ist zu umfangreich, um in einer einzigen Diplomarbeit ausreichend behandelt werden zu können. Aus diesem Grund beschäftigen sich zwei Diplomarbeiten mit diesem Thema. Die Arbeit von Stephanie Wist [41] deckt die Themen um das verteilte System und das Anwenderprogramm ab. Die vorliegende Arbeit kümmert sich um das Framework. Die Schnittpunkte zwischen diesen beiden Teilen werden jeweils aus der entsprechenden Perspektive in beiden Arbeiten behandelt.

Diese Arbeit ist folgendermaßen gegliedert: das zweite Kapitel erklärt zuerst, welche alternativen Lösungen existieren und warum diese den Anforderungen nicht gerecht werden. Außerdem liefert dieses Kapitel kurze Einführungen in relevante Themengebiete und verweist dabei ggf. auf weiterführende Literatur.

Der Systementwurf in Kapitel 3 diskutiert, wie die Anforderungen umgesetzt und die dabei entstehenden Probleme gelöst werden. Das nachfolgende Kapitel 4 über die Implementierung beschäftigt sich damit, wie die Lösungen des Entwurfs tatsächlich umgesetzt werden.

Trotz der Aufteilung in zwei Diplomarbeiten werden nicht alle Fragen geklärt, die sich bei der Diskussion um Lethe ergeben. Kapitel 6 listet diese offenen Fragen auf und bietet einen Ausblick auf eventuelle Folgearbeiten. Die Zusammenfassung in Kapitel 7 rundet die Arbeit ab.

1.2 Die Anforderungen

Wie schon erwähnt muss das Framework vor allem die Erstellung von Sicherungspunkten und die Migration von Simulationen ermöglichen. Darüber hinaus ist die Wiederverwendbarkeit von Implementierungen eine zentrale Anforderung.

Auf Grund der langen Simulationslaufzeiten ist außerdem die Ausführungsgeschwindigkeit der Simulationen von großem Interesse. Des Weiteren muss die Programmierschnitt-

stelle zum einen mächtig genug sein, um alle Anwendungsfälle abzudecken und zum anderen intuitiv bedienbar sein. Letzteres impliziert vor allem, dass der Entwickler keinen Code schreiben muss, dessen Zweck und Hintergründe nicht offensichtlich sind. Außerdem müssen überall Vorgaben existieren, die zu einem sinvollen Verhalten der Laufzeitumgebung führen. Damit muss der Entwickler nur das bedenken, was für seinen Fall von Relevanz ist.

2 Grundlagen und Stand der Technik

Dieses Kapitel besteht aus zwei Teilen. Zuerst klärt Abschnitt 2.1, welche Alternativen zu Lethe existieren und an welchen Stellen es sich von ihnen abgrenzt. Die restlichen Abschnitte bieten jeweils eine kurze Einführung in die Themen, die für die Diskussionen um Lethe relevant sind.

2.1 Simulationsprogramme

Ein weitverbreitetes und sehr beliebtes Simulationsprogramm ist Matlab [24]. Dabei ist es im eigentlichen Sinn kein Simulationsprogramm, sondern eine allgemeine Plattform für numerische Berechnungen. Gründe für die Beliebtheit und die weite Verbreitung sind sicherlich die mächtige Matlab Programmiersprache und die sehr umfangreichen Bibliotheken. Damit ist es möglich, einfach und in kurzer Entwicklungszeit komplexe Berechnungen durchzuführen. Mit etwas Programmieraufwand lassen sich diese Berechnungen schließlich zu einer Simulation verknüpfen.

Ein großes Problem, das sich dem Einsatz von Matlab in den Weg stellt, ist der hohe Preis für eine Lizenz. Je nach Lizenztyp kostet das Basispaket bis zu 1900 US Dollar. Dazu kommen weitere Kosten für Zusatzpakete, die zwischen $ 400 und $ 5000 liegen (Stand 10. April 2006). Für den, der sich diese Ausgaben sparen möchte, besteht die Möglichkeit, auf eine freie Alternative wie Scilab [33] oder mit eingeschränktem Funktionsumfang auch GNU Octave [28] zurückzugreifen.

Die Unterstützung für verteiltes und paralleles Rechnen muss bei Matlab mit Hilfe von Zusatzpaketen nachgerüstet werden. Die meisten dieser Pakete verwenden eine Implementierung des “Message Passing Interface” (MPI [27]) oder die “Parallel Virtual Machine” (PVM [31]). Beides bedeutet, dass der Entwickler die Algorithmen zur Verteilung von Hand implementieren muss. Letzteres zwingt sogar zum Einsatz einer homogenen Hardwarelandschaft. Diesen Nachteil bringen auch die Pakete mit, die eine transparente Verteilung bieten, da sie einen Cluster wie z.B. Beowulf [1] voraussetzen. Scilab unterstützt die PVM von Haus aus. Für GNU Octave existiert ein MPI basierter Klon namens Parallel Octave. Beide bringen jeweils die oben genannten Nachteile mit sich.

Matlab und seine Alternativen bringen keine Unterstützung zum Migrieren von Simulationen oder zum Erstellen von Sicherungspunkten mit. Auch die Verteilung der Rechenkapazitäten im Netzwerk und die Durchsetzung von Interessen verschiedener Benutzergruppen muss mit Mitteln der jeweiligen Systeme von Hand gemacht werden. Da so etwas von einer Plattform für numerische Berechnungen auch nicht erwartet wird, ist das nicht verwunderlich. Dennoch ist der Einsatz dieser Programme in einem Netzwerk deshalb unkomfortabel und fehlerträchtig.

Wer dem oben erwähnten Programmieraufwand zur Erstellung einer Simulation entgehen möchte, greift zu einem auf Simulationen spezialisierten Programm, wie z.B. GoldSim [15]. Über eine graphische Benutzeroberfläche kann der Anwender dort seine Simulation zusammenstellen und die einzelnen Komponenten konfigurieren. Dabei kann er auf eine umfangreiche Menge von fertigen Kompontenten zurück greifen, so dass er keine Programmierkenntnisse braucht. Am Ende der Simulation kann er sich die Ergebnisse graphisch anzeigen lassen und mit Hilfe von mitgelieferten Werkzeugen analysieren. Das ist alles sehr komfortabel. Inwiefern ein Anwender selbst neue Kompontenten entwickeln kann, ist allerdings unklar. Die oben zitierte Internetseite schweigt sich darüber aus. Die Lizenz für eine kommerzielle Nutzung von GoldSim kostet $ 3950 für das Basispaket. Für akademische Zwecke existiert eine kostenlose Version mit einem etwas eingeschränkten Funktionsumfang. Für Zusatzpakete kommen allerdings auf jeden Fall Kosten zwischen $ 1000 und $ 9000 hinzu (Stand 10. April 2006).

Bekannte Alternativen zu GoldSim sind z.B. MLDesigner [26], das "CoCentric Systems Studio" [7] und das freie Ptolemy [30]. Obwohl die drei Programme jeweils einen etwas anderen Einsatzzweck haben, ergeben sich für den Kanalcodierer dieselben Probleme. So ist keines der Programme auf den Einsatz in der Kanalcodierung spezialisiert. Das bedeutet, dass der Kanalcodierer eine geeignete Abbildung seines Problems auf das Modell des Simulationsprogrammes finden muss. Das kann mitunter sehr aufwendig und frustrierend sein. Dazu kommt, dass keines der Programme die in der Einführung aufgelisteten Probleme beim Einsatz in Netzwerken löst. Der Anwender muss selbständig verteilen und die einzelnen Simulationen verwalten. Simulationen migrieren oder Sicherungspunkte erstellen ist überhaupt nicht möglich.

2.2 Simulationsalgorithmen

Auf dem Gebiet der parallelen und verteilten Simulationen sind vor allem die Publikationen von Richard M. Fujimoto maßgeblich. Sein Buch "Parallel and Distributed Simulation Systems" [13] behandelt Ideen zur Beschleunigung von Simulationen durch paralleles und verteiltes Rechnen. In den ersten Kapiteln liefert er außerdem einen Überblick über bekannte Architekturen für Simulationen. Die folgenden Definitionen und Erläuterungen beziehen sich alle auf dieses Buch.

Eine Simulation ist, abstrakt beschrieben, eine Menge von Teilnehmern, die nach gewissen Regeln untereinander Informationen austauschen. Dieser Informationsaustausch findet ausschließlich durch Ereignisse statt, die von Teilnehmern ausgelöst werden und bei einem empfangenden Teilnehmer Reaktionen hervorrufen. Ein solches Ereignis besitzt, außer seinen simulationsspezifischen Informationen und einer Kennung für den Empfänger, einen Zeitpunkt, zu dem es behandelt werden soll. Gestartet wird die Simulation durch ein initiales Ereignis. Da nicht pro Teilnehmer eine Recheneinheit zur Verfügung steht, muss ein sogenannter Scheduler entscheiden, wann welcher Teilnehmer an der Reihe ist. Außerdem muss er den Austausch der Ereignisse verwalten. Im Wesentlich existieren zwei Ansätze hierfür: der getaktete und der ereignisbasierte Simulator.

Die Architekturen der getakteten Simulatoren sind sehr schlicht. Für jede Zeiteinheit der Simulation existiert eine Runde, in der nach dem Round-Robin Prinzip jeder Teil-

nehmer einmal an der Reihe ist. Die dabei ausgelösten Ereignisse werden vom Scheduler eingesammelt. Zu Beginn der nächsten Runde stehen alle Ereignisse, deren Behandlungszeitpunkt gekommen ist, ihren Empfängern zur Verfügung. Der Nachteil dieser Architketur ist, dass es jedesmal, wenn ein Teilnehmer für keines der Ereignisse einer Runde der Empfänger ist, zu einer Leer-Operation kommt. Je nach Simulation ist die Anzahl der Leer-Operationen hoch und damit die Laufzeitverschwendung signifikant.

Der ereignisbasierte Simulator arbeitet mit einer Warteschlange für Ereignisse, die nach dem Behandlungszeitpunkt sortiert sind. Der Scheduler nimmt nacheinander die obersten Ereignisse aus der Schlange und gibt sie dem Empfänger zur Behandlung. Der kann daraufhin neue Ereignisse erzeugen, die wiederum in die Warteschlange aufgenommen werden. Der Effizienzvorteil dieser Architektur begründet sich dabei auf zwei Eigenschaften. Erstens werden keine unnötigen Leer-Operationen verursacht. Zweitens muss die Simulationszeit nicht linear verlaufen. So wird z.B. das oberste Ereignis in der sortierten Warteschlange immer sofort behandelt, auch wenn der gewünschte Zeitpunkt der Behandlung in der Zukunft liegt. Die Simulationszeit macht in diesem Fall einfach einen entsprechend großen Sprung.

Eine Möglichkeit, die Laufzeit zu verkürzen, ist, die Berechnungen zu verteilen, um sie parallel in mehreren Recheneinheiten zu erledigen. Die dadurch entstehenden Nebenläufigkeiten können allerdings zu Konflikten führen. Ein Konflikt ist z.B. wenn zwei Ereignisse auf Grund der nicht synchronen Simulationszeiten der einzelnen Recheneinheiten von einem Empfänger in der falschen Reihenfolge behandelt werden. Um diesem Problem zu begegnen, existieren im Wesentlichen zwei Ansätze: die pessimistischen und die optimistischen Verfahren[1].

Die in der Fachwelt als konservativ bezeichneten, pessimistischen Verfahren gehen davon aus, dass ein Konflikt wahrscheinlich ist und investieren deshalb in die Vermeidung. Dazu verwendet das Scheduling die Kausalitätskette der Ereignisse, um voneinander abhängige Ereignisse ausschließlich in der richtigen Reihenfolge zu behandeln. Kausal unabhängige Ereignisse können hingegen parallel behandelt werden. Um die Korrektheit des Verfahrens zu garantieren, muss das Scheduling im Zweifelsfall annehmen, dass eine kausale Abhängigkeit zwischen zwei Ereignissen existiert und kann ihre Bearbeitung damit nicht parallelisieren. Die Effizienz dieses Verfahrens hängt deshalb direkt von der Vollständigkeit der ermittelten Kausalitäten ab. Da die Kausalitäten eine Eigenschaft der Simulationslogik sind, können sie nicht automatisiert ermittelt werden. Vielmehr braucht das Scheduling dazu Metainformationen vom Benutzer. Die Effizienz von pessimistischen Verfahren hängt also im Endeffekt davon ab, wie viele Kausalitäten der Benutzer erkennt und angibt.

Die optimistischen Verfahren, die in der Fachwelt auch als timewarp-Verfahren bezeichnet werden, gehen stattdessen davon aus, dass ein Konflikt unwahrscheinlich ist. Daher sparen sie sich den Aufwand für die Vermeidung und nehmen dafür höhere Kosten für die Behandlung eines Konfliktes in Kauf. Für den Ablauf einer Simulation heißt das, dass wenn ein Ereignis verspätet eintrifft, alle zu früh behandelten Ereignisse rück-

[1] Diese Bezeichnungen sind für Simulatoren eigentlich unüblich. Trotzdem wurden sie an dieser Stelle gewählt, da sich die Kernidee dahinter in anderen Gebieten der Informatik, wie z.B. den Datenbanken [39], wiederfindet und diese Begriffe dort verwendet werden.

gängig gemacht werden müssen. Anschließend werden diese Ereignisse zusammen mit dem verspäteten Ereignis in der richtigen Reihenfolge neu behandelt. Der Nachteil dieser Verfahren ist offensichtlich die große Anzahl an Informationen, die der Scheduler speichern muss, um behandelte Ereignisse vollständig rückgängig machen und erneut behandeln lassen zu können.

Ist zwischen den einzelnen Recheneinheiten keine Kommunikation während der Simulation erforderlich, spricht die Fachwelt von einem “embarrassingly parallel” Algorithmus, weil dieser Fall keine Herausforderung darstellt.

2.3 Multitasking

In einer Simulation in der Kanalcodierung laufen mehrere Algorithmen gleichzeitig und unabhängig voneinander ab. Es findet also per Definition ein Multitasking statt. Dieser Abschnitt diskutiert, welche Möglichkeiten des Multitaskings existieren und welche davon die schnellste ist. Zur Klärung der hier verwendeten Begriffe kann z.B. [38] konsultiert werden.

Eine Form des Multitaskings ist die Verwendung von mehreren Prozessen. Das ist die typische Architektur moderner Betriebssysteme, um mehrere Programme gleichzeitig und unabhängig voneinander auszuführen. Im Falle einer Simulation existiert also pro Algorithmus ein Prozess. Da die einzelnen Prozesse allerdings in voneinander getrennten Adressräumen liegen, muss zur Kommunikation das Betriebssystem verwendet werden. Neben den Kontextwechseln zwischen den Prozessen, die auf Grund des Wechsels des Adressraumes bereits teuer sind, müssen also noch Kontextwechsel ins Betriebssystem gemacht werden. Dieser Ansatz ist also sehr langsam.

Eine bessere Möglichkeit ist, jeden Algorithmus in einem eigenen Thread laufen zu lassen. Zwei Threads laufen, im Gegensatz zu zwei Prozessen, im selben Adressraum. Der Kontextwechsel zwischen ihnen ist also billiger und für die Kommunikation wird das Betriebssytem nicht gebraucht. Dennoch wird weiterhin das Scheduling vom Betriebssytem erledigt. Doch auch dieser Ansatz kostet unnötig Laufzeit:

Erstens erfordert die Nebenläufigkeit eine Synchronisation an den Übergabestellen. Dazu muss ein kritischer Bereich mit Hilfe einer Semaphore, eines Mutexes oder eines Monitors angelegt werden. Je nach System erfordert das ggf. einen Aufruf ans Betriebssystem, die Erstellung von Speicherbarrieren oder die Entleerung der CPU-Caches. Da während einer Simulation viele Daten zwischen den Algorithmen ausgetauscht werden, werden die kritischen Bereiche sehr oft betreten, so dass der Aufwand dafür die Laufzeit spürbar erhöht.

Zweitens verursacht das präemptive Multitasking des Betriebssystems viel mehr Kontextwechsel als nötig. Je länger jeder Thread seine Arbeit erledigt bis er unterbrochen wird, desto seltener muss zwischen zwei Threads umgeschaltet werden. Anstatt also nach einem Zeitscheibenverfahren präemptiv umzuschalten, kann eine programmeigene Logik ihr Wissen über die Aufgaben der Threads und die Übergabepunkte ausnutzen, um seltener und zu den günstigen Zeitpunkten kooperativ umzuschalten. Neben einer deutlichen Senkung der Anzahl der Kontextwechsel fallen zusätzlich die kritischen Bereiche weg, weil es dann keine echte Nebenläufigkeit mehr gibt.

Für diesen Ansatz werden sogenannte User Threads benötigt. Im Unterschied zu Kernel Threads weiß das Betriebssystem nichts über deren Existenz, womit die Verantwortung für das Scheduling in der Hand des Programms liegt. Für den Entwickler eines Algorithmus ändert sich dabei verglichen mit dem präemptiven Fall nichts, wenn die Übergabe in einem Bibliotheksaufruf der Laufzeitumgebung stattfindet. In beiden Fällen ist die Startfunktion des Threads die wohldefinierte Umgebung für lokale Daten des Algorithmus.

Die "GNU Portable Threads" Bibliothek (PTH [11]) ist für diesen Zweck eine gute Wahl. Dem Autor Ralf Engelschall ist es nämlich gelungen [10], kooperatives Multitasking ohne die Verwendung von plattformspezifischem Assembler und ausschließlich unter Verwendung von POSIX-konformen (ISO 9945) Schnittstellen zu implementieren.

Allerdings haben Messungen im Rahmen dieser Arbeit ergeben (Anhang A.1), dass ein Kontextwechel zwischen zwei User Threads ungefähr so viel Laufzeit kostet, wie 100 nicht-optimierte Funktionsaufrufe. Auch wenn es im Rahmen dieser Arbeit gelungen ist, die Anzahl der Kontextwechsel gegenüber der aktuellen PTH Implementierung zu halbieren (Anhang A.2), bleibt die Verwendung der PTH teuer. Andere Bibliotheken sind vielleicht schneller, verwenden dafür aber nicht-portablen Assembler Code.

Möchte man also die Laufzeit einer Simulation niedrig halten, muss man dem Entwickler ein wenig Komfort absprechen. Anstatt eine eigene ausführende Einheit zu haben, muss er seinen Algorithmus als Behandlungsfunktion von geeignet definierten Ereignissen formulieren, damit das kooperative Multitasking durch einen zentralen Verteiler erledigt werden kann.

Da dieser Ansatz sehr schnell ist, ist er typisch für Simulationen. Es handelt sich um den in Abschnitt 2.2 diskutierten ereignisbasierten Simulator.

2.4 Hypergraphen

Die Graphentheorie ist eine alte und allgemein bekannte Disziplin der Mathematik. Dementsprechend exsitiert sehr viel Literatur zu diesem Thema. Allerdings ist die Definition eines Hypergraphen nicht allgemein bekannt. Deshalb erläutert dieser Abschnitt sehr knapp die zentralen Eigenschaften, die in späteren Kapiteln von Relevanz sind.

Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten. Jede Kante verbindet immer genau zwei Knoten. Bei einem gerichteten Graphen besitzen die Kanten eine Richtung. Der ungerichtete Graph ist ein Spezialfall des gerichteten Graphen, da eine ungerichtete Kante zwei gerichteten Kanten mit entgegengesetzer Richtung entspricht. Jede beliebige Teilmenge von Knoten mit dazugehörigen Kanten wird als Teilgraph bezeichnet.

Ein Hypergraph ist eine Verallgemeinerung eines Graphen. Anstatt Kanten besitzt ein Hypergraph sogenannte Hyperkanten, die mehr als nur zwei Knoten miteinander verbinden können. Ist die Hyperkante gerichtet, zerfällt die Menge der verbundenen Knoten in die Menge der Anfangskonten und die Menge der Endknoten. Gängige Eigenschaften von Graphen, wie z.B. Zyklen, sind für Hypergraphen analog definiert.

Jeder ungerichtete Hypergraph lässt sich auf einen einfachen Graphen abbilden, indem ein neuer Knoten eingeführt wird, der mit allen Knoten der Hyperkante durch eine

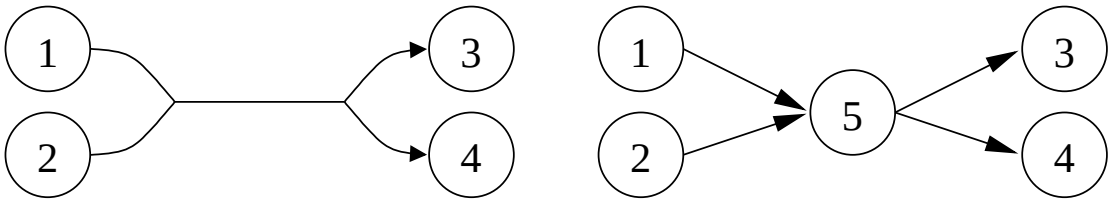


Abbildung 2.1: Abbildung eines gerichteten Hypergraphen auf einen gerichteten Graphen

einfache Kante verbunden ist. Unter Berücksichtigung der Richtungen lässt sich auf die gleiche Weise jeder gerichtete Hypergraph auf einen gerichteten Graphen abbilden. Abbildung 2.1 verdeutlicht das.

Ein Hypergraph besitzt demzufolge dieselbe Mächtigkeit wie ein Graph. Er benötigt lediglich keine Extraknoten, um die gemeinsame Konnektivität einer beliebigen Menge von Knoten darzustellen.

2.5 C++

Wie Abschnitt 4.1 diskutiert, muss ein Entwickler seine Algorithmen in C++ implementieren. Dieser Abschnitt liefert deshalb einen kurzen Überblick über diese Sprache. Als weiterführende Literatur wird das Buch “Die C++ Programmiersprache” des C++ Erfinders Bjarne Stroustrup ([35]) empfohlen.

Der Grund für die Entwicklung von C++ war, dass alle in den 70ern bekannten Programmiersprachen entweder sehr ineffizient oder sehr rudimentär waren. Stroustrup erkannte das damals, als all seine Versuche, eine Software für ein Projekt zu schreiben, scheiterten. Der Grund war entweder, dass die gewählte Sprache keine Werkzeuge mit sich brachte, um die Komplexität der Implementierung in den Griff zu bekommen, oder, dass das Programm am Ende zu langsam war. Damit er nie wieder solchen Problemen ausgesetzt sein würde, machte er sich daran, eine neue Sprache zu entwickeln [36].

Stroustrup wollte, dass mit seiner Arbeit real existierende Probleme kurzfristig gelöst werden können. Da die von Ken Thompson und Dennis Ritchie entwickelte Sprache C [4] damals weit verbreitet war, hat er diese als Ausgangsbasis verwendet und um neue Konzepte erweitert. Diese Konzepte hat er größtenteils von Simula-67 [5] und später von Algol [37] übernommen. Mittlerweile ist aus dieser Arbeit der ISO-C++ Standard geworden (ISO/IEC 14882:1998), der auf dem C90 Standard von C aufbaut (ISO/IEC 9899:1990).

C++ ist also, bis auf wenige Details, eine Obermenge von C. Die Kenntnisse über die Sprache C sind so weit verbreitet, dass eine Einführung im Rahmen dieser Arbeit nicht nötig ist. Die oben erwähnten neuen Konzepte werden allerdings für das Verständnis von Kapitel 4 benötigt und sollten deshalb bei Bedarf angeeignet werden. Deshalb sind sie im folgenden mit einem jeweiligen Verweis auf das jeweils entsprechende Kapitel im Buch von Bjarne Stroustrup [35] aufgelistet.

Klassen Das sind die zentralen Strukturen der objektorentierten Programmierung (Kapitel 10).

Vererbung Ein wichtiger Mechanismus der objektorentierten Programmierung, der die Spezialisierung oder Verallgemeneinerung von Klassen ermöglicht. Man spricht in diesem Zusammenhang auch von einer *Ableitung* (Kapitel 12).

Templates Ein zur Vererbung orthogonales Konzept der generischen Programmierung. Es ermöglicht die datentypunabhängige Formulierung von Algorithmen (Kapitel 13).

Ausnahmen Ein Konzept zur Fehlerbehandlung. Es ermöglicht eine Trennung des Ortes, an dem ein Fehler auftritt, vom Ort, an dem der Fehler behandelt wird (Kapitel 14).

überladene Operatoren Ein wichtiger Teil der Gleichstellung von benutzerdefinierten und eingebauten Typen. Durch die Verwendung von gängigen Operatoren ist es möglich, intuitive Schnittstellen zu definieren (Kapitel 11).

Namensräume Ermöglichen eine hierarchische Definition von Bezeichnern. Dadurch lassen sich Namenskonflikte zwischen verschiedenen Teilen der Software vermeiden. Ein qualifizierter Bezeichner gibt durch den Bereichsoperator `::` an, in welchem Namensraum er definiert ist. Beginnt er mit dem Bereichsoperator, so ist die Adressierung absolut zur Wurzel der Hierarchie zu verstehen. (Kapitel 8).

Neben den Vorteilen hat C++ aber auch viele Nachteile von C geerbt. Beispiele hierfür sind die komplexe Syntax, Lücken in der Sprachdefinition, variable Parameterlisten, der Präprozessor und das Linker Modell. Außerdem fehlen der Sprache wichtige Konzepte, die in modernen Programmiersprachen zum Alltag gehören. Dazu gehört ein Import-Konzept, das dynamische Nachladen von Code, eine umfangreiche Standardbibliothek, Unterstützung für nebenläufiges Programmieren und - umstritten, aber dennoch zu nennen - eine automatische Garbage Collection.

Gängige Compiler unterstützen das Sprachmittel “export” ([35], Kapitel 13.7) nicht. Dadurch ist man gezwungen, die Definition und die Deklaration von Template-Klassen (und Funktionen) in einer Datei zu machen. Die Modularisierung des Programms ist damit aufgebrochen. Einer der Nachteile, die sich daraus ergeben, ist, dass in jeder Übersetzungseinheit, die die Deklaration der Template-Klasse braucht, alle Abhängigkeiten der Implementierung dieser Klasse mit eingefügt werden.

2.6 Numerische Bibliotheken

Die Algorithmen, die in der Kanalcodierung auftreten, führen häufig numerische Berechnungen durch. Für diesen Zweck gibt es schon eine Reihe von Bibliotheken.

Am weitesten verbreitet sind dabei die “Basic Linear Algebra Subprograms” (BLAS [6], [22]). Diese in Fortran [19] geschriebene, hoch optimierte Bibliothek ist die Basis für viele erweiterte numerische Bibliotheken, wie z.B. das “Linear Algebra Package“ (LAPACK [9]) oder die ”GNU Scientific Library“ (GSL [23]).

Eine alternative Möglichkeit ist die Verwendung der Matlab Bibliotheken, die sehr umfangreich sind. Implementiert man die Verwendung dieser Bibliotheken in der komfortablen Matlab Hochsprache, kann man anschließend mit Hilfe eines Matlab-nach-C-Compilers die Methoden in ein in C oder C++ geschriebenes Programm einbinden.

Außer einer Einschränkung, die in Abschnitt 3.1.9 erläutert wird, spielt die tatsächlich verwendete numerische Bibliothek im Grunde keine Rolle [2]. Es ist allerdings wichtig zu wissen, dass es verschiedene Bibliotheken gibt und dass viele Entwickler ihre Favoriten haben. Um einen möglichst großen Anwenderkreis ansprechen zu können, unterstützt Lethe die Verwendung beliebiger Bibliotheken, sogar innerhalb einer Simulation. Für die Beispielimplementierungen wurde die GSL verwendet, da sie weit verbreitet und mächtig ist.

[2] Es ist allerdings zu beachten, dass Lethe unter der GPL steht (vgl. Abschnitt 6.3)

3 Systementwurf

Der erste Teil des Entwurfes erarbeitet zum einen ein Modell der Simulation und zum anderen eine Architektur des Frameworks. Der zweite Teil des Entwurfes beschäftigt sich mit den Teilnehmern des verteilten Systems, die Inhalt dieser Arbeit sind. Er diskutiert, welche Teilnehmer welche Aufgaben haben und welche Schnittstellen zur Kommunikation zwischen ihnen existieren.

Warum Lethe ein verteiltes System ist und es die Dienste der "Internet Communication Engine" (ICE [21]) verwendet, wird in [41] diskutiert. Auch die Frage, wie der Entwickler und der Benutzer die von Lethe benötigten Metainformationen zu den Algorithmen und den Simulationen angeben kann, ist Thema von [41]. Wichtig für diese Arbeit hier ist nur, dass es dafür einen Weg gibt und dass alle benötigten Informationen den Teilnehmern in Form einer Simulationsbeschreibung zur Verfügung gestellt werden.

3.1 Die Simulation

Die Diskussionen in diesem Abschnitt sind vor allem von zwei Anforderungen geprägt. Erstens wird eine für Kanalcodierer spezialisierte Lösung geschaffen. Dadurch sind an einigen Stellen Optimierungen möglich, die eine bessere Laufzeit oder einen geringeren Aufwand gegenüber allgemeingehaltenen Ansätzen mit sich bringen.

Zweitens soll es aus Gründen der Robustheit und einer flexiblen Verteilung der Simulationen möglich sein, von einer laufenden Simulation einen Zwischenzustand zu erheben, der jederzeit ggf. auf einem anderen Computer fortgesetzt werden kann. Das muss in einer plattformunabhängigen Art und Weise geschehen, damit zum einen beliebige und zum anderen heterogene Hardwarelandschaften unterstützt werden.

Die Diskussionen in diesem Abschnitt werden zur besseren Verständlichkeit von einem einfachen Beispiel begleitet. Abschnitt 3.1.1 stellt den Ausgangspunkt vor. Spätere Abschnitte modifizieren dieses Beispiel stellenweise, um spezielle Aspekte des Systementwurfs veranschaulichen zu können.

Das für die Simulationen verwendete Modell muss zugleich einfach und mächtig sein. Abschnitt 3.1.2 erklärt, warum der Hypergraph (vgl. Abschnitt 2.4) diese beiden Eigenschaften besitzt. In seinen Knoten befinden sich dabei die von den Entwicklern definierten Algorithmen der Kanalcodierung, die über die Kanten miteinander kommunizieren.

In welcher Weise das Multitasking zwischen den Knoten funktioniert, diskutiert Abschnitt 3.1.3. Insbesondere stellt sich dabei heraus, dass ein in der gesichteten Literatur nicht erwähnter Algorithmus für das Scheduling von Simulationen im Fall von Lethe den geringsten Aufwand und die geringsten Laufzeitkosten bringt.

Die Frage nach der Terminierung des Schedulings ist nicht trivial. Deshalb diskutiert Abschnitt 3.1.4, wie das Ende einer Simulation definiert ist. Dazu führt er die Begriffe

des *Beobachters* und des *Protokollierers* ein. Beides sind optionale Eigenschaften eines Knotens, die zu einer Sonderbehandlung durch das Scheduling führen. Die Menge der Beobachter bestimmt über das Ende der Simulation. Für die Protokollierer ist gewährleistet, dass sie bis zum Ende an der Simulation teilnehmen.

Eine Möglichkeit, die Simulationsgeschwindigkeit zu steigern, ist es, die Simulation zu verteilen um sie parallel durchzuführen. Abschnitt 3.1.5 führt dazu den Begriff der *Runden* ein, und diskutiert warum im Falle von Lethe ausschließlich an den Grenzen dieser voneinander unabhängigen Teile effizient parallelisiert werden kann.

Abschnitt 3.1.6 beschäftigt sich anschließend damit, warum Zyklen im Graphen benötigt werden und wie das Scheduling mit dadurch möglichen Verklemmungen umgeht. Da sich bei Zyklen die Frage nach Möglichkeiten zur Vorinitialisierung von Kanten stellt, führt Abschnitt 3.1.7 die Begriffe der *Phasen* und der *Ports* ein. Die zentrale Überlegung dabei ist, dass während einer Phase jeweils nur ein Teilgraph aktiv ist.

Die Vorinitialisierung von Kanten führt auf das Problem von mehreren Schreibern hin. Abschnitt 3.1.8 erklärt, warum dieses Problem direkt von den Eigenschaften des Schedulings herrührt und wie man es lösen kann.

Abschnitt 3.1.9 diskutiert, wie mit Hilfe einer ICE Funktionalität die plattformunabhängige Erhebung von Zwischenzuständen ermöglicht wird. Außerdem erklärt er, auf was ein Entwickler dabei achten muss.

Um einen Überblick über die Interaktionen der Knoten mit der Laufzeitumgebung zu erhalten, erklärt Abschnitt 3.1.10 den Lebenszyklus eines Knotens. Einen Überblick über die vom Benutzer benötigten Metainformationen über einen Knoten bietet Abschnitt 3.1.11.

3.1.1 Das Fallbeispiel

Der Kern des Fallbeispieles ist ein AWGN Kanal, der die übertragenen Daten additiv mit einem weißen gaussverteilten Rauschen überlagert. Ein Faltungscode ([2], Kapitel 8) soll die Fehlerwahrscheinlichkeit der Übertragung unter einen gewissen Wert drücken. Die zu übertragenen Daten stammen aus einer Quelle, die gewisse, hier nicht näher erläuterte statistische Eigenschaften hat. Der Kanalcodierer ist daran interessiert, welche Fehlerwahrscheinlichkeiten sich für verschiedene Rauschleistungen des Kanals ergeben. Ermittelt wird diese Wahrscheinlichkeit von einer Senke, die die Daten der Quelle mit den Daten des Decoders vergleicht. Das ist der Ausgangspunkt der folgenden Diskussionen.

3.1.2 Das Modell

Eine gebräuchliche Darstellung für Simulationen ist das sogenannte Blockschaltbild. Dabei werden einfache Rechtecke und Pfeile verwendet, wobei ein Rechteck z.B. für den Encoder oder den Kanal, also einem eigenständigen Algorithmus steht. Ein Pfeil bedeutet, dass die damit verbundenen Rechtecke über diesen Weg in der eingezeichneten Richtung kommunizieren. Diese Darstellung ist intuitiv und erfahrungsgemäß mächtig genug, um die allermeisten Fälle abbilden zu können.

Es liegt also nahe, den gerichteten Graphen als Modell für Simulationen in der Kanalcodierung zu verwenden. Er besteht aus einer Menge von Knoten und einer Menge von Kanten mit einer Richtung, die jeweils genau zwei Knoten miteinander verbinden. Die Idee ist, dass die Algorithmen der Simulation in den Knoten ablaufen und die Kanten eine Infrastruktur zur Kommunikation der Knoten untereinander aufbauen. Jede Kante besitzt folglich einen Puffer, der die Ausgaben des schreibenden Knoten dem lesenden als Eingabe zur Verfügung stellt. Abbildung 3.1 zeigt den Simulationsgraphen für das Fallbeispiel.

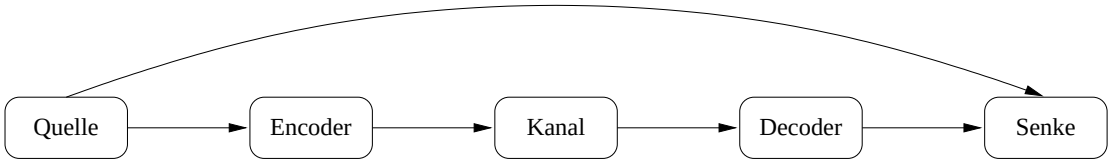


Abbildung 3.1: Ein Simulationsgraph

Die Algorithmen in den Knoten sind also beliebige Abbildungen von k Eingängen auf n Ausgänge, wobei entweder k oder n null sein kann. Im Fallbeispiel nimmt der Algorithmus im Kanal fortlaufend Zahlen vom Encoder entgegen, addiert gaussverteilte Zufallszahlen dazu und stellt das Ergebnis dem Decoder zur Vefügung. Der Kanal verwendet sinnvollerweise jedes Mal eine andere Zufallszahl. Damit das möglich ist, müssen die Knoten zustandsbehaftet sein.

Die obige Beschreibung des Kanals ist nicht präzise. Denn eigentlich weiß der Kanal nicht, welche anderen Knoten sich an seinem Eingang und seinem Ausgang befinden. Aus Gründen der Wiederverwendbarkeit der Algorithmen sollte das auch prinzipiell so sein. Im Graphen aus Abbildung 3.1 ist dieses Prinzip für die Quelle allerdings verletzt. So hängt die Anzahl der ausgehenden Kanten davon ab, ob eine Senke vorhanden ist oder nicht. Um die Logik der Verwendung innerhalb eines Graphen aus den Algorithmen in den Knoten fern halten zu können, muss das Modell zu einem Hypergraphen (vgl. Abschnitt 2.4) erweitert werden. Eine Kante kann dann mehrere Leser haben. Dabei muss die Laufzeitumgebung garantieren, dass alle Leser die gleichen Daten sehen. Abbildung 3.2 zeigt, wie der Graph aus Abbildung 3.1 korrekt aussehen muss.

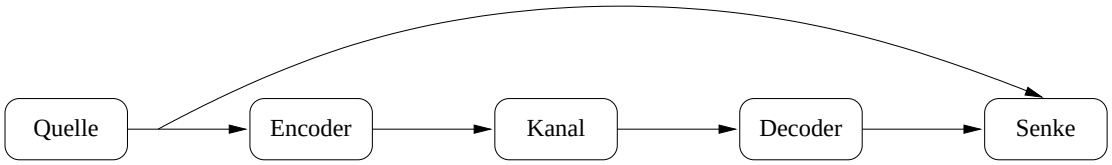


Abbildung 3.2: Der Hypergraph des Fallbeispieles

Jeder Knoten kann eine Konfiguration und Einstellungen besitzen. Die Konfiguration des Encoders aus dem Fallbeispiel ist die Generatormatrix, die dessen Realisierung als Schieberegister beschreibt. Damit kann derselbe Knoten für die Simulation beliebiger

Faltungscodes verwendet werden. Die Einstellung des Kanals aus dem Fallbeispiel ist die Rauschleistung, die er verwenden soll. In mehreren Runden derselben Simulation kann somit der gesuchte Verlauf des Fehlerwahrscheinlichkeit ermittelt werden.

3.1.3 Das Scheduling

Die Laufzeitumgebung muss das gleichzeitige und voneinander unabhängige Ablaufen der einzelnen Algorithmen in den Knoten ermöglichen. Abschnitt 2.3 diskutiert die bekannten Möglichkeiten für dieses Multitasking und kommt zu dem Schluß, dass ein ereignisbasierter Simulator die schnellste dieser Möglichkeiten ist.

Die Definition von geeigneten Ereignissen ist allerdings nicht trivial, denn nach der gängigen Definition (vgl. Abschnitt 2.2) existiert neben den Ereignissen kein zweiter Informationsweg zwischen den Teilnehmern einer Simulation. Im Fall von Simulationen in der Kanalcodierung führt das aber zu Problemen. So kann es z.B. sein, dass eine Quelle einzelne Zahlen erzeugt, der Encoder aber immer große Tupel von Zahlen lesen möchte. Bei Blockcodes [2] ist dies z.B. der Fall. Es ist unklar, wie in einem solchen Fall die Ereignisse zu definieren sind. Entweder muss die Quelle Teilereignisse erzeugen oder der Encoder muss mehrere Ereignisse auf einmal bearbeiten. Alternativ kann ein Hilfsknoten eingeführt werden, der aus mehreren Ereignissen der Quelle ein Ereignis für den Encoder erzeugt. Diese Lösung erhöht allerdings den ohnehin schon erheblichen Aufwand, der für die Erzeugung, die Verwaltung in der Warteschlange und die Zustellung von Ereignissen entsteht.

Da sich Lethe ausschließlich auf Simulation in der Kanalcodierung konzentriert, kann es diesen Problemen aus dem Weg gehen und einen in der Literatur für Simulationen untypischen Ansatz verwenden. Die Idee von Ereignissen und einem zentralen Entscheider wird aufgegeben. Statt dessen orientiert sich das Scheduling am Aufbau des Hypergraphen. Die Daten werden über die Kanten transportiert und die Leser und Schreiber bei Bedarf aufgerufen. Die Schedulingentscheidungen werden dabei dezentral in den Kanten getroffen, weshalb ein solches Ereignis im Folgenden als ein *impliziter Aufruf* bezeichnet wird.

Das Scheduling funktioniert also folgendermaßen:

- Der Scheduler wählt einen beliebigen Knoten und ruft ihn. Ein solcher Knoten wird im Folgenden als *initialer Knoten* bezeichnet.
- Möchte ein Knoten aus einer Kante mehr Daten lesen als darin gepuffert sind, so ruft die Kante ihren Schreiber so oft, bis er genug Daten erzeugt hat, um den Aufruf des Lesers zu befriedigen.
- Möchte ein Knoten in eine Kante mehr Daten schreiben als der Puffer dieser Kante Platz besitzt, so geht die Kante analog vor. Sie ruft wiederholt einen ihrer Leser bis genug Platz im Puffer entstanden ist, um den Aufruf des Schreibers durchführen zu können. Dabei wählt sie jedesmal denjenigen Leser, der bisher am wenigsten gelesen hat und daher am meisten Platz im Puffer verbraucht.
- Ist der Aufruf des initialen Knotens beendet, so beginnt der Scheduler von vorne.

Die einzige Aufgabe des Schedulers besteht darin, fortlaufend einen initialen Knoten zu wählen. Es lässt sich deshalb darüber streiten, ob er überhaupt noch als Scheduler zu bezeichnen ist. Aus Mangel an alternativen gebräuchlichen Bezeichnungen wird dieser Algorithmus in dieser Arbeit trotzdem weiterhin als Scheduling bezeichnet.

Die Folge von impliziten Aufrufen baut einen Wartestack auf, der wieder abgebaut wird, wenn die Bedingungen für die wartenden Lese- oder Schreibaufrufe erfüllt sind. Die Höhe des Stacks ist durch die Anzahl der Knoten beschränkt, da zirkuläre Warteabhängigkeiten verboten sind und die Laufzeitumgebung dieses Verbot durchsetzt (vgl. Abschnitt 3.1.6).

Da ein Knoten bei einem wartenden Schreibaufruf frühestens dann weiter machen kann, wenn der Leser mit dem meisten Verbrauch seine Daten gelesen hat, wählt die Kante immer diesen Leser aus. Damit wird die Zeit, die der Aufruf des initialen Knotens benötigt, minimal gehalten. Das ist wichtig, um kurze Antwortzeiten bei der Erhebung eines Zwischenzustandes zu ermöglichen (vgl. Abschnitt 3.1.9).

Es muss noch geklärt werden, ob der Aufruf eines initialen Knotens unter Garantie terminiert. Es ist nämlich denkbar, dass sich die Folge von impliziten Aufrufen ständig in einem Teilgraphen hin und her bewegt. Um das zu verhindern ist folgendes hinreichend:

Jeder Knoten muss garantieren, dass jede Kante immer wieder innerhalb von endlich vielen Aufrufen bedient wird. Dann ist nämlich sichergestellt, dass jeder wartende Lese- oder Schreibaufruf irgendwann erledigt werden kann, falls der implizit gerufene Knoten immer wieder aufgerufen wird.

Durch das Verbot von zirkulären Warteabhängigkeiten muss die Folge von impliziten Aufrufen immer irgendwann einen Knoten erreichen, der seine Arbeit erledigen kann. Spätestens an diesem Punkt wächst der Wartestack nicht weiter und wird bis zu einem Knoten abgebaut, für den die Bedingungen seines wartenden Lese- oder Schreibaufrufes noch nicht erfüllt sind. Falls ein Knoten dazwischen einen weiteren Lese- oder Schreibaufruf macht, wird der Abbau des Wartestacks enstprechend früher unterbrochen. Auf jeden Fall wächst der Wartestack anschließend wieder. Insgesamt ist so garantiert, dass alle wartenden Knoten immer wieder aufgerufen werden, was zu zeigen war.

Da ein Entwickler eine ein- oder ausgehende Kante nicht ohne Grund vorsieht, ist die oben genannte hinreichende Bedingnung im Normalfall erfüllt. Allerdings liegt das tatsächlich in der Hand des Entwicklers. Ist die Bedingung nicht erfüllt, so kann das höchstens die Laufzeitumgebung mit Hilfe einer Heuristik erkennen, weil die zuverlässige Erkennung sehr wahrscheinlich ein nicht entscheidbares Problem darstellt. Allerdings wurde das im Rahmen dieser Arbeit nicht nachgewiesen. Auch wird die angesprochene Idee einer Heuristik nicht weiter verfolgt.

Das vorgestellte Scheduling führt dazu, dass fortlaufend alle Knoten bei Bedarf aufgerufen werden, um Daten zu erzeugen, zu konsumieren oder zu verarbeiten. Mit jedem Aufruf eines initialen Knotens durch den Scheduler läuft die Simulation ein Stück weiter. Abschnitt 3.1.4 erläutert, wann der Scheduler damit aufhört, so dass die Simulation zu ihrem Ende kommt.

Der Nachteil des Schedulingalgorithmus ist, dass die Aufrufreihenfolge der Knoten erstens von der Wahl der initialen Knoten, zweitens von den Größen der Puffer in den Kanten und drittens vom Verhalten der einzelnen Knoten abhängt. Die Wahl der initialen Knoten ist, wie Abschnitt 3.1.4 erläutert, nicht beliebig und vom Anwender

beeinflussbar. Die Größe der Puffer ist, wie Abschnitt 4.2.2 erklärt, auch relativ einfach zu bestimmen. Doch die Komplexität des Zusammenspiels dieser Faktoren und vor allem die Tatsache, dass das Verhalten eines Knotens eventuell unbekannt ist, wenn man nicht dessen Entwickler ist, legen nahe, trotz ihrer Determiniertheit keine Annahmen über die Aufrufreihenfolge zu machen.

Eine direkte Konsequenz aus diesem Nachteil ist, dass die Menge der verwendeten Algorithmen beschränkt ist auf die Klasse der *stationären Algorithmen*. So können z.B. Interrupts nicht abgebildet werden. Eine weitere direkte Konsequenz ist, dass eine Kante maximal einen Schreiber haben kann (vgl. Abschnitt 3.1.8).

3.1.4 Das Ende einer Simulation

Ein scheinbar trivialer aber im Detail doch komplexer Sachverhalt ist die Frage, wann eine Simulation zu Ende ist. Ein denkbarer Anwendungsfall ist, dass die Simulation beendet werden soll, wenn die Senke eine bestimmte Anzahl von Vergleichen gemacht hat. Allerdings könnte es auch sein, dass die Simulation frühzeitig beenden soll, wenn z.B. die Senke feststellt, dass die Performance des simulierten Codes sehr schlecht ist. Es sind weiterhin Anwendungsfälle denkbar, bei denen nur Teile der Simulation beenden während die anderen weiter laufen sollen. Das ist z.B. der Fall, wenn der Benutzer mehr als eine Senke in der Simulation platziert und die Senken verschiedene Kriterien für das Ende verwenden.

Es ist einem Knoten deshalb möglich, der Laufzeitumgebung mitzuteilen, dass für ihn das Ende der Simulation erreicht wurde, woraufhin sie ihn aus dem Graphen entfernt. Außerdem kann der Benutzer eine beliebige Teilmenge von Knoten als relevant für die Entscheidung über das Ende der Simulation markieren. Diese Knoten werden als *Beobachter* bezeichnet. Wurden alle Beobachter aus dem Graphen entfernt ist die Simulation zu Ende. Diese Vorgehensweise ist zugleich einfach zu verstehen und sehr flexibel.

Eine Frage, die sich hier sofort stellt, ist, wie der restliche Graph auf fehlende Knoten reagiert. Dazu existieren die folgenden drei Regeln:

1. Falls ein Lesezugriff auf eine Kante zu einem impliziten Aufruf des Schreibers führen würde und der Schreiber dieser Kante bereits beendet hat, so entfernt die Laufzeitumgebung den Knoten aus dem Graphen, falls der Knoten diesen Sonderfall nicht behandelt.

2. Falls ein Schreibzugriff auf eine Kante zu einem impliziten Aufruf eines Lesers führen würde, und alle Leser der ausgehenden Kanten des Schreibers bereits beendet haben, so entfernt die Laufzeitumgebung den Knoten aus dem Graphen, falls der Knoten diesen Sonderfall nicht behandelt.

3. Falls ein Schreibzugriff auf eine Kante zu einem impliziten Aufruf eines Lesers führen würde, der Leser dieser Kante bereits beendet hat und mindestens ein Leser einer ausgehenden Kante dieses Knotens noch nicht entfernt wurde, so bleibt der Knoten im Graphen. Die geschriebenen Daten werden nachträglich verworfen. Die Laufzeitumgebung kann den Schreiber optional über diesen Sonderfall informieren.

Wenn ein Knoten der Laufzeitumgebung mitteilt, dass für ihn das Ende der Simulation erreicht wurde, dann beendet er *explizit*. Wird ein Knoten allerdings auf Grund der obigen Regeln aus dem Graphen entfernt, beendet er *implizit*.

Der Grund für die erste Regel ist, dass sich auf diese Weise Entscheidungen über das Ende der Simulation automatisch im Graphen fortpflanzen. Außerdem stellt sich die Frage, was in so einem Fall sonst mit dem Knoten geschehen soll. Schließlich ist es ihm aus Mangel an eingehenden Daten nicht mehr möglich, seiner Aufgabe weiterhin nach zu kommen. Sollte es im Einzelfall nötig sein, dass der Knoten trotzdem weitermacht, so kann er seine Entfernung verhindern, indem er den Sonderfall behandelt (vgl. Abschnitt 4.3.2).

Ähnliches gilt für die zweite Regel. Der Grund hierfür ist, dass die Arbeit eines Knotens im Normallfall nicht mehr nötig ist, wenn kein anderer Knoten mehr an seinen Daten interessiert ist. Auch hier kann der Knoten seine Entfernung falls nötig verhindern.

So lange es noch Leser gibt, die auf die Daten eines Knotens angewiesen sein könnten, ist es wichtig, dass dieser Knoten normal weiterarbeitet. Deshalb muss es für ihn in so einem Fall transparent sein, ob die Kante, in die er schreibt, noch einen Leser besitzt oder nicht. Das ist der Grund für die dritte Regel.

Hier stellt sich noch die Frage, warum ein Knoten nicht sofort aus dem Graphen entfernt wird, wenn der letzte Leser beendet. Der Grund dafür ist, dass es für die Initialisierung von Kanten (vgl. Abschnitt 3.1.7) nötig ist, dass ein Knoten in eine Kante schreiben kann, zu der noch kein Leser existiert.

Um zu garantieren, dass die Simulation terminiert, ist es hinreichend, wenn jeder Beobachter vom Benutzer ein erreichbares Abbruchkriterium erhalten hat und der Scheduler jeden verbliebenen Beobachter regelmäßig als initialen Knoten wählt. Letzteres garantiert, dass jeder Beobachter die Möglichkeit erhält, sein Abbruchkriterium zu erreichen, auch wenn der Graph durch die Entfernung von Knoten in Teilgraphen zerfällt. Ein isolierter Beobachter, der durch das implizite Scheduling nicht mehr erreicht werden kann, würde andernfalls zu einer nicht terminierenden Simulation führen. Da ein Beobachter auch implizit beenden kann, ist die obige Bedingung übrigens keine notwendige Bedingung für die Terminierung der Simulation.

Wählt der Scheduler die initialen Knoten ausschließlich aus der Menge der verbliebenen Beobachter, kann das zu Teilgraphen führen, die durch das implizite Scheduling nicht mehr erreicht werden. Allerdings wird so eine Situation vom Scheduler nicht explizit erkannt, so dass die Knoten des unereichbaren Teilgraphen nicht beendet werden.

Ist in so einem unereichbaren Teilgraphen ein Knoten, der mit protokollieren soll, führt das zu einem unvollständigen Protokoll. Um dies zu verhindern hätte der Benutzer zwar die Möglichkeit, diesen Knoten als Beobachter zu markieren. Allerdings müsste dieser Knoten dann auch mitentscheiden, wann die Simulation zu Ende ist. Wenn er das nicht kann oder nicht soll, gibt es ein Problem. Um das zu lösen bietet Lethe eine weitere Klasse von besonders markierten Knoten, die sogenannten *Protokollierer*. Diese werden vom Scheduler neben den Beobachtern regelmäßig als initiale Knoten gewählt, spielen aber keine Rolle bei der Festlegung des Endes der Simulation.

3.1.5 Verteilung

Wie der letzte Absatz in Abschnitt 3.1.2 erklärt, besteht eine Simulation im Allgemeinen aus einer Menge von Runden, die sich untereinander in den Einstellungen der Knoten unterscheiden. Dieser Begriff einer *Runde* besitzt im Kontext von Lethe genau diese Bedeutung. Jeder Knoten bekommt am Anfang jeder Runde seine vom Benutzer vorgegebenen Einstellungen. Das Aussehen dieser Einstellungen wird vom jeweiligen Entwickler festgelegt (vgl. [41]).

Die zentrale Eigenschaft von Runden ist, dass sie alle voneinander unabhängig sind. Damit eignen sie sich ideal für eine parallele Berechnung, denn die Verteilung der Runden einer Simulation auf mehrere Computer oder mehrere Prozessoren ist "embarrassingly parallel" (vgl. Abschnitt 2.2). Nach welcher Strategie Lethe das macht ist Thema von [41].

Eine automatisierte zeitliche Verteilung mit einer feineren Granularität wird von Lethe aus zwei Gründen nicht unterstützt. Erstens weiß die Laufzeitumgebung nicht, wann eine Runde zu Ende ist (vgl. Abschnitt 3.1.4). Damit ist es unklar, wie eine Runde aufgeteilt werden kann. Zweitens kann es für das Ergebnis einer Simulation eine Rolle spielen, ob eine Runde am Stück oder in mehrere Teile aufgespaltet simuliert wird. Denn im letzten Fall besitzt ein Knoten am Anfang eines Teils einen initialen Zustand, der vom Zustand am Ende des vorherigen Teils abweichen kann. Nur wenn für alle Knoten gilt, dass diese Abweichung das Ergebnis nicht verfälscht, kann eine Runde parallelisiert werden. Da die automatisierte Suche nach einer geeigneten Aufspaltung vermutlich ein nicht entscheidbares Problem ist, verfolgt Lethe diesen Ansatz nicht weiter. Ob die Vermutung zutrifft wird im Rahmen dieser Arbeit allerdings nicht entschieden. Eine verbleibende Möglichkeit ist, dass der Benutzer eine Aufspaltung manuell durchführt (vgl. Abschnitt 6.4.3).

Denkbar ist auch eine räumliche Verteilung der Knoten auf mehrere Computer. Dann müssen alle Daten, die über Kanten von einem Teilgraphen in den anderen fließen, über das lokale Netzwerk transportiert werden. Da während einer Simulation sehr viele Daten verarbeitet werden, ist der Aufwand dafür erheblich. Die positive Wirkung der Verteilung auf die Laufzeit wird also durch den steigenden Aufwand für den Datentransport relativiert. Dazu kommt, dass in den meisten Simulationen der größte Rechenaufwand im Decoder steckt. Eine erhebliche Beschleunigung durch die Verteilung der anderen Knoten ist also in den meisten Fällen nicht zu erwarten. Da dieser Ansatz also nicht gewinnbringend zu sein scheint, wird er in dieser Arbeit nicht weiter verfolgt.

Es bleibt also dabei, dass ausschließlich an Rundengrenzen parallelisisert wird. Auf Grund der Unabhängigkeit der Runden ist keine Kommunikation zwischen den einzelnen Teilsimulationen nötig. Deshalb ist das Verhältnis zwischen den Laufzeiten einer vollständig parallelisierten Simulation und der dazugehörigen nicht parallelisierten Simulation antiproportional zur Anzahl der Runden. Das ist in den meisten Fällen bereits ein erheblicher Gewinn. Dieser Sachverhalt ist natürlich trivial. Nicht umsonst bemühen sich Kanalcodierer um eine gleichzeitige Berechnung der unabhängigen Teile ihrer Simulationen. Die Leistung von Lethe an dieser Stelle ist allerdings, dass die Verteilung automatisiert gemacht wird. Das ist komfortabler, weniger fehleranfällig und erlaubt es dem Benutzer, sich auf die eigentlichen Probleme zu konzentrieren.

3.1.6 Feedback-Kanten

Bei den *ARQ* Verfahren [3] passt der Encoder die Menge der Redundanz an die aktuellen Eigenschaften eines veränderlichen Kanals an. Dazu muss der Decoder dem Encoder mitteilen, wie gut oder wie schlecht die Decodierung der Daten gelingt. In der Summe wird dadurch bei gleichbleibender Fehlerwahrscheinlichkeit weniger Redundanz übertragen. Abbildung 3.3 zeigt, dass die zugehörige Feedback-Kante einen Zyklus im Hypergraphen dieser Simulation erzeugt. Dieser Abschnitt diskutiert, welche Konsequenzen sich daraus ergeben.

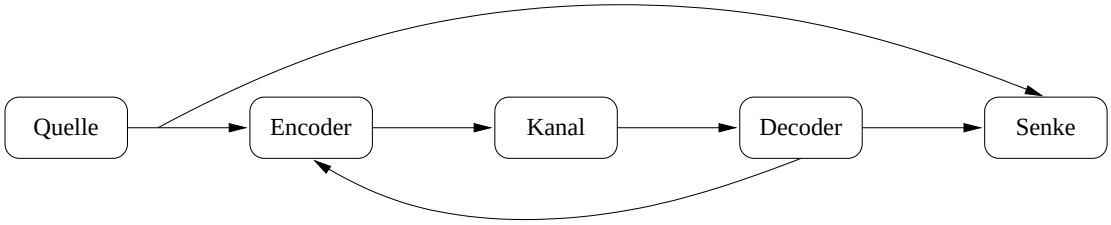


Abbildung 3.3: Der Hypergraph eines ARQ Verfahrens

Abschnitt 3.1.3 braucht für die Korrektheit des Schedulings ein Verbot von zirkulären Warteabhängigkeiten. Der Grund dafür ist, dass es andernfalls zu einer unendlichen Folge von impliziten Aufrufen entlang des Zykluses kommen kann. Die Simulation würde damit nie terminieren.

Im Beispiel der ARQ Verfahren kann es folgendermaßen zu einer zirkulären Warteabhängigkeit kommen: der Decoder braucht zuerst Daten vom Kanal. Erst danach kann er die für den Encoder notwendigen Daten bestimmen und sie zur Verfügung stellen. Allerdings braucht der Kanal seinerseits Daten vom Encoder. Möchte jetzt der Encoder erst Feedback-Daten vom Decoder, bevor er entscheidet, wie er die Daten von der Quelle codiert, wird wiederum der Decoder gerufen. Dieser braucht aber wieder erst Daten vom Kanal, um arbeiten zu können. Auf diese Weise werden die drei Knoten unendlich lange im Kreis herum aufgerufen.

Um das zu verhindern, muss sich z.B. der Encoder anders verhalten. So kann er beim allerersten Aufruf initiale Feedback-Daten annehmen, anstatt sie über die Feedback-Kante zu lesen. Abschnitt 3.1.7 beschreibt eine Alternative dazu.

Es ist zu vermuten, dass eine zirkuläre Warteabhängigkeit immer eine Verklemmung darstellt, deren Ursache im Verhalten der Knoten, dem Aufbau der Simulation oder einer ungenügenden Vorinitialisierung zu suchen ist. Deshalb handelt es sich dabei um einen Logikfehler in der Simulation und nicht um eine Einschränkung der Möglichkeiten von Lethe. Die Erfahrung wird zeigen müssen, ob diese Vermutung uneingeschränkt zutrifft.

Für Lethe bedeutet das, dass es das Verbot von zirkulären Warteabhängigkeiten durchsetzt, um robust auf solche Fälle reagieren zu können. Es erkennt, wenn ein Knoten, der bereits im Wartestack vorhanden ist, implizit aufgerufen wird und bricht daraufhin die Simulation ab. Der Benutzer wird anschließend darüber informiert und erhält für die Fehlersuche eine Auflistung der wartenden Knoten.

3.1.7 Phasen

Wie Abschnitt 3.1.6 erklärt, ist es zur Vermeidung von Verklemmungen manchmal nötig, dass einer der beteiligten Knoten eine Sonderbehandlung für seinen ersten Aufruf vorsieht. Das erhöht allerdings die Komplexität der Knoten und behindert deren Wiederverwendbarkeit.

Deshalb muss eine Möglichkeit vorgesehen werden, Feedback-Kanten mit Daten zu initialisieren, bevor die eigentliche Simulation beginnt. Da allerdings nicht auszuschließen ist, dass die Initialisierungswerte algorithmisch bestimmt werden müssen, braucht der Benutzer mehr als nur eine Möglichkeit, die Werte anzugeben. Am besten wäre eine Lösung, die möglichst viele Anwendungsfälle abdeckt, ohne dabei kompliziert zu sein.

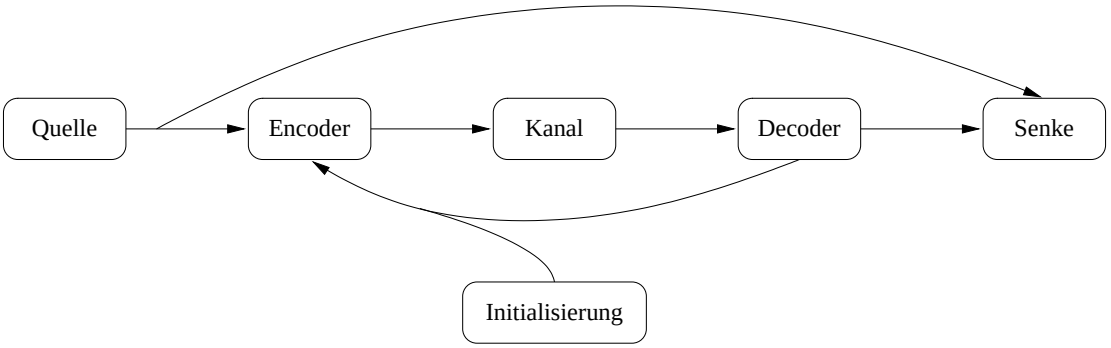


Abbildung 3.4: Initialisierung einer Feedback-Kante

Lethe kennt zu diesem Zweck das Konzept von *Phasen*. Eine Runde einer Simulation besteht aus beliebig vielen Phasen. Jeder Knoten kann pro Phase aktiv oder inaktiv für jede ein- und ausgehende Kante sein. Damit dafür ein handlicher Begriff existiert, definiert Lethe einen *Port* als das Verbindungsstück zwischen einem Knoten und einer Kante. Aktiv oder inaktiv ist damit pro Phase eine Eigenschaft der Ports, die der Benutzer festlegt.

Beim Übergang von einer Phase in die nächste bleiben die Daten in den Kanten erhalten. Die Simulation in Abbildung 3.4 besteht z.B. aus zwei Phasen. In der ersten Phase ist nur der eine Port des Initialisierungsknotens aktiv. In der zweiten Phase ist dieser Port dann inaktiv und dafür alle anderen aktiv. Damit ist die Kante vom Decoder zum Encoder am Anfang der zweiten Phase vorinitialisiert.

Beendet ein Knoten während einer Phase, sind alle seine Ports ab diesem Zeitpunkt implizit inaktiv. Wenn ein Port einer Kante inaktiv ist, kann ein anderer Port derselben Kante durchaus aktiv sein. Falls keiner der verbundenen Knoten die Kante verwendet, ist sie damit implizit inaktiv. Eine inaktive Kante behält ihre enthaltenen Daten einfach, bis sie wieder aktiv wird oder die Simulation zu Ende ist.

Versucht ein Knoten auf einen inaktiven Port zuzugreifen, bricht die Laufzeitumgebung die Simulation ab und informiert den Benutzer. Dieser muss anschließend untersuchen, warum sich der Knoten anders verhält als von ihm erwartet.

Im Laufe einer Simulation werden alle Phasen nacheinander ausgeführt. Für das Ende einer Phase gelten die in Abschnitt 3.1.4 diskutierten Regeln. Vor dem Beginn der nächsten Phase nimmt die Laufzeitumgebung alle Knoten wieder in den Graphen auf und gibt dem Scheduler eine neue benutzerdefinierte Menge von Beobachtern und Protokollierern. Anschließend aktiviert oder deaktiviert die Laufzeitumgebung die Ports und startet den Scheduler. Die Simulation ist schließlich zu Ende, wenn die letzte Phase zu Ende ist.

Es gibt mehrere Möglichkeiten, festzulegen, welche Daten einer Kante ein aktiver Leser zu Beginn einer Phase sieht. Die einfachste davon ist, dass die Kante für ihn leer ist. Falls es für den Leser keine Rolle spielt, was in der vorhergehenden Phase geschehen ist, ist es wichtig, dass diese Möglichkeit existiert. Deshalb kann der Benutzer für jeden eingehenden Port pro Phase angeben, ob die angeschlossene Kante zu Beginn dieser Phase für den Leser leer oder gefüllt sein soll. Falls es für einen Knoten, wie im Beispiel aus Abbildung 3.4 für den Encoder, von Relevanz ist, dass in der vorherigen Phase Daten in die eingehende Kante geschrieben wurden, wird der Benutzer für diesen Knoten letzteres angeben. Einfach gesprochen, stehen dem Knoten in dem Fall die Daten aus der vorhergehenden Phase zur Verfügung. Doch welche Daten das sinnvollerweise sind, ist nicht trivial zu entscheiden.

Die folgenden Regeln legen fest, welche Daten ein Leser zu sehen bekommt, wenn der Benutzer festlegt, dass er zu Beginn einer Phase eine gefüllte Kante sehen soll.

1. Wenn ein Leser am Ende der alten Phase aktiv war und zu Beginn der neuen Phase aktiv ist, dann gehen für ihn die beiden Phasen nahtlos ineinander über. Die Daten, die er am Ende der alten Phase gesehen hat, sind die Daten, die er zu Beginn der neuen Phase sieht.

2. Ansonsten ist dieser Leser neu an der Kante. Dann hängt die Entscheidung für ihn von den anderen Lesern der Kante ab.

   a) Falls es andere Leser gibt, die am Ende der alten Phase aktiv waren, dann ist von ihnen der Leser ausschlaggebend, für den die Kante am Ende der alten Phase die meisten Daten gepuffert hatte. Diese Daten sind diejenigen, die der neue Leser zu Beginn der neuen Phase sieht.

   b) Ansonsten ist der Leser ausschlaggebend, der als letztes beendet hat. Die Daten, die die Kante zum Zeitpunkt seines Beendens für ihn gepuffert hatte, sind diejenigen, die dem neuen Leser zu Beginn der neuen Phase zur Verfügung stehen. Hinzu kommen die Daten, die der Schreiber nach dem Beenden des letzten Lesers ggf. in die Kante eingefügt hat.

Die erste Regel entspricht dem, was der Benutzer normalerweise erwartet. Die Regel 2a) bewirkt, dass ein neuer Leser so viel Daten wie möglich zur Verfügung hat. Die Festlegung der Regel 2b) hat den Sinn, dass nach dem Beenden des letzten Lesers der Schreiber die Kante noch maximal füllen kann.

Es ist bei Regel 2 möglich, dass der neue Leser die ersten Daten, die er in der neuen Phase liest, in der alten schon einmal gelesen hat. Das hängt davon ab, wann er in der alten Phase beendet hat und wie sich die anderen Leser danach verhalten haben. Wenn

die Menge der Daten, die der neue Leser am Anfang der neuen Phase sieht, vorhersagbar sein muss, dann muss dieser Leser während der alten Phase entweder gar nicht oder als letzter beenden. Denn dann gilt entweder Regel 1 oder Regel 2b, so dass er auf jeden Fall nahtlos weitermachen kann. Die Tatsache, dass das Scheduling keine Garantie über die Aufrufreihenfolge gibt (vgl. Abschnitt 3.1.3), kann allerdings dazu führen, dass sich vorab nicht bestimmen lässt, ob der Leser eventuell implizit beendet wird, bevor der letzte Beobachter beendet. Ob das für viele Anwendungsfälle problematisch ist, wird die Erfahrung mit Lethe zeigen müssen.

In Regel 2b) können zwischen dem Zeitpunkt des Beendens des letzten Lesers und dem Zeitpunkt, zu dem der neue Leser aktiv wird, beliebig viele Phasen liegen, denn eine Kante kann über mehrere Phasen hinweg inaktiv sein. Der Fall, in dem zu Beginn der ersten Phase einer Runde alle Leser inaktiv sind, wird behandelt, als ob genau ein Leser zu Beginn dieser ersten Phase aktiv war und sofort beendet hat. Das Beispiel aus Abbildung 3.4 ist so ein Fall. Der Encoder bekommt dadurch genau den Inhalt, den der Initialisierungsknoten geschrieben hat. Dabei muss allerdings sichergestellt sein, dass der Puffer der Kante groß genug ist, um alle Daten aufnehmen zu können. Da nur der Benutzer weiß, wie viel Daten das sind, bietet Lethe ihm die Möglichkeit, für jede Kante eine Mindestgröße des Puffers anzugeben. Die tatsächliche Größe des Puffers berechnet sich dann aus dieser und weiteren Metainformationen (vgl. Abschnitt 4.2.2).

3.1.8 Mehrere Schreiber

Am Ende von Abschnitt 3.1.3 wird erwähnt, dass eine Kante maximal einen Schreiber haben kann. Dennoch ist die Abbildung 3.4, in der eine Kante zwei Schreiber besitzt, wie im vorherigen Abschnitt ausgeführt, korrekt. Genauer formuliert muss die Aussage aus Abschnitt 3.1.3 nämlich heißen, dass an einer Kante pro Phase maximal ein Schreiber aktiv sein darf.

Der Grund für diese Beschränkung ist die Tatsache, dass das Scheduling keine Garantie über die Aufrufreihenfolge der Knoten gibt. Deshalb ist es bei mehreren aktiven Schreibern an einer Kante unklar, was ein Leser dieser Kante zu sehen bekommt. Er kann nicht davon ausgehen, dass die Daten aus der Kante abwechselnd von den jeweiligen Schreibern stammen. Die Schreiber müssten deshalb Metainformationen über die Herkunft der Daten hinzufügen. Allerdings ist diese Vermischung von Daten und Simulationslogik nicht wünschenswert und vermutlich eine potentielle Problemquelle. Deshalb verbietet Lethe schlichtweg die Verwendung von mehreren aktiven Schreibern an einer Kante.

Es ist allerdings möglich, eine Kante mit n Schreibern auf n Kanten mit je einem Schreiber abzubilden (vgl. Abschnitt 2.4). Damit lässt sich das Problem mit mehreren Schreibern umgehen, indem der Entwickler den Leser mit n eingehenden Kanten versieht. Falls die Anzahl der Schreiber für den Leser transparent sein muss, kann der Entwickler einen Knoten entwerfen, der zwischen den Schreibern und dem Leser ein geeignetes Multiplexing vornimmt.

3.1.9 Zwischenzustände

Eines der besonderen Merkmale von Lethe ist die Möglichkeit, von einer Simulation Sicherungspunkte zu erstellen und Simulationen auf andere Computer zu migrieren. Da der einzige Unterschied zwischen diesen beiden Aktionen darin besteht, dass die Simulation bei einer Migration auf einem anderen Computer fortgesetzt wird, verwenden beide Funktionalitäten denselben Mechanismums.

Die Kernidee ist, dass jeder Teil einer Simulation, der zustandsbehaftet ist, diesen Zustand in geeigneter Form vollständig persistiert. Wird dieser Zustand zu späterer Zeit und evtl. an einem anderen Ort von allen Teilen wieder exakt angenommen, so läuft die Simulation ab diesem Zeitpunkt genau so ab, wie sie abgelaufen wäre, wenn man nie einen Zwischenzustand erfasst hätte.

Ermöglicht wird das sehr einfach durch ICE [21]. Der Mechanismus zum plattformunabhängigen Austausch von Objekten wird hier lediglich auf eine andere Art verwendet. Lethe lässt die Objekte von der ICE Laufzeitumgebung serialisieren und speichert die so erzeugten Datenströme, anstatt sie z.B. über das lokale Netz zu senden. Zur Rekonstruktion der Zustände nimmt Lethe diese gespeicherten Daten wieder her, um sie von der ICE Laufzeitumgebung deserialisieren zu lassen und die Zustandsobjekte zu erhalten. Die Plattformunabhängigkeit dieser Vorgehensweise ergibt sich dabei direkt aus der Plattformunabhängigkeit von ICE.

Eine Simulation ist damit nichts anderes, als ein Wirt für die Manipulation von Zustandsobjekten. Existieren auf zwei Computern zwei gleiche Wirte, kann die Manipulation der Zustandsobjekte beliebig oft abwechselnd in beiden Wirten fortgesetzt werden. Zwei Wirte sind genau dann gleich, wenn sie aus derselben Simulationsbeschreibung erzeugt wurden (vgl. Abschnitt 3.2.3).

Der Entwickler muss beachten, dass er auch Zustände aus ggf. verwendeten Bibliotheken mit berücksichtigt. Bietet eine zustandsbehaftete Bibliothek keine Möglichkeit, ihre Zustände zu lesen und zu schreiben, ist sie für den Einsatz innerhalb von Lethe schlecht geeignet. In diesem Fall muss der Entwickler entscheiden, ob sich die Tatsache, dass nach einem Umzug diese Zustände wieder einen initialen Wert annehmen, auf das Simulationsergebnis auswirkt. Falls es eine Möglichkeit gibt, diese aber nicht plattformunabhängig ist, bedeutet ihre Verwendung, dass der Einsatz der entsprechenden Implementierungen auf homogene Netzwerke beschränkt ist.

Es liegt in der Hand des Entwicklers, was der Zustand eines Knotens alles umfassen kann. Die Arbeit von Stephanie Wist [41] behandelt, wie der Entwickler die Zustandsmenge, die übrigens durchaus leer sein kann, spezifiziert und wie sichergestellt wird, dass die ICE Laufzeitumgebung damit umgehen kann.

Bei der Erhebung eines Zwischenzustandes werden auch die Zustände der Kanten erfasst. Dazu gehören insbesondere die im Puffer enthaltenen Daten. Zum Zustand des Schedulers gehören die aktuellen Mengen der Beobachter, der Protokollierer und der bereits beendeten Knoten und die Nummer der aktuellen Phase.

Um zu garantieren, dass es für die Ergebnisse keine Rolle spielt, ob eine Simulation einmal persistiert und wieder fortgesetzt wurde oder nicht, muss das Scheduling bei einer Fortsetzung an exakt derselben Stelle weitermachen. Da es keine portable Möglichkeit gibt, den Wartestack des Schedulings auf einen anderen Computer zu übertragen, bleibt

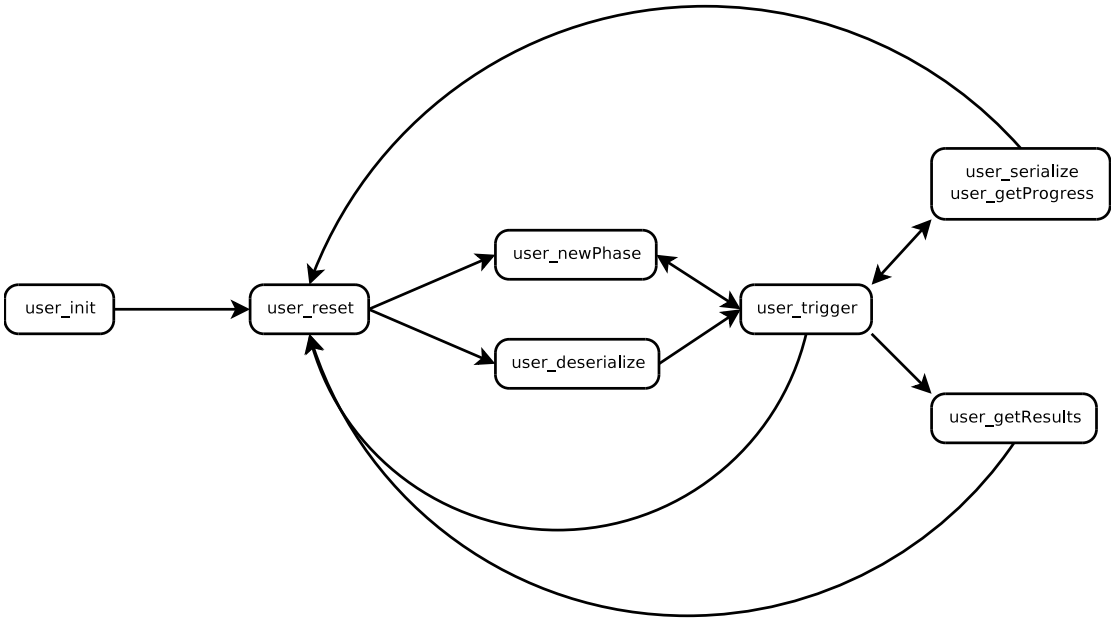


Abbildung 3.5: Interaktion eines Knotens mit der Laufzeitumgebung

als einzige einfache Möglichkeit, mit der Erfassung des Zwischenzustandes zu warten, bis der Aufruf des initialen Knotens beendet und der Wartestack damit leer ist. Das Scheduling (vgl. Abschnitt 3.1.3) garantiert, dass der Wartestack immer wieder leer wird und sorgt außerdem dafür, dass die Wartezeit darauf minimal ist.

3.1.10 Der Lebenszyklus eines Knotens

Der Lebenszyklus eines Knotens ist eine Folge von verschiedenen Interaktionen mit der Laufzeitumgebung. Zu jeder Interaktionsmöglichkeit gehört in der Implementierung eine Methode, deren Namen in Abbildung 3.5 vorweggreifend schon genannt werden.

Im Folgenden ist zusammenfassend und ergänzend die Menge der Interaktionen eines Knotens mit der Laufzeitumgebung aufgelistet. Die Namen der zugehörigen Methoden der Implementierung sind dabei hervorgehoben.

Initialisierung (`user_init`) Zu Beginn einer Simulation erhält jeder Knoten bei seiner Initialisierung eine benutzerdefinierte Konfiguration (vgl. Abschnitt 3.1.2, letzter Absatz).

Beginn einer neuen Runde (`user_reset`) Jeder Knoten bekommt zu Beginn einer neuen Runde seine Einstellungen. (vgl. Abschnitt 3.1.5).

Beginn einer neuen Phase (`user_newPhase`) Jeder Knoten wird über den Beginn einer neuen Phase informiert, damit er sich geeignet verhalten kann (vgl. Abschnitt 3.1.7).

Aufruf (`user_trigger`) Jeder Knoten wird entweder explizit vom Scheduler oder implizit von einer Kante wiederholt gerufen. Während eines Aufrufes kann der Knoten entweder explizit oder implizit beenden (vgl. Abschnitt 3.1.3).

Erfassen des Zustandes (`user_serialize`) Bei der Erfassung eines Zwischenzustandes muss jeder Knoten seinen aktuellen Zustand erfassen und ihn der Laufzeitumgebung zur Verfügung stellen (vgl. Abschnitt 3.1.9)

Laden des Zustandes (`user_deserialize`) Bei der Wiederaufnahme eines Zwischenzustandes muss jeder Knoten den zuvor erfassten Zustand wieder annehmen (vgl. Abschnitt 3.1.9).

Erfragen des Fortschrittes (`user_getProgress`) Es kann sein, dass der Benutzer einen Überblick über den Fortschritt der Simulation haben möchte. Neben der Nummer der aktuell laufenden Phase ist es dabei von Interesse, wie weit diese Phase fortgeschritten ist. Da ihr Ende der Laufzeitumgebung vorab nicht bekannt ist, kann Lethe diese Information nicht selbständig liefern. Deshalb fragt es alle Knoten, wie viel Prozent der Phase ihrer Meinung nach schon erledigt sind. Wenn ein Knoten keine Aussage über den Fortschritt machen kann, ist 100% die passende Antwort, da die Laufzeitumgebung von allen Angaben das Minimum wählt. Welche Aussagekraft der Fortschritt für den Benuzter besitzt, liegt an der jeweiligen Simulation. Verhalten sich die Knoten passend, ist der Verlauf des Fortschrittes z.B. linear, so dass eine Abschätzung der Gesamtlaufzeit möglich ist.

Einsammeln der Ergebnisse (`user_getResult`) Am Ende der Simulation wird jeder Knoten aufgefordert, aus den protokollierten Informationen ein Ergebnis zu ermitteln und es der Laufzeitumgebung zu übergeben. Falls ein Knoten nichts protokolliert hat, kann er eine leere Menge übergeben.

Zerstören eines Knotens (`user_finish`) Am Ende der Simulation wird jeder Knoten gebeten, evtl. belegte Ressourcen wieder frei zu geben. Anschließend werden alle Knoten und die Laufzeitumgebung zerstört. Diese Methode kann zu jedem Zeitpunkt aufgerufen werden und ist deshalb in Abbildung 3.5 zur Erhaltung der Übersichtlichkeit nicht eingezeichnet.

In welcher Reihenfolge die Interaktionen tatsächlich statt finden, hängt vom Manager (vgl. Abschnitt 3.2) ab. Dieser bestimmt nämlich entweder automatisch oder gesteuert durch den Benutzer, was mit den Simulationen geschehen soll. Abbildung 3.5 zeigt, welche Reihenfolgen der Interaktionen dabei allgemein möglich sind.

3.1.11 Die Simulationsbeschreibung

Lethe braucht für die Simulation eine ganze Reihe von Metainformationen vom Benutzer. Diese werden im Folgenden zusammengefasst:

- die Menge der beteiligten Knoten

- den Hypergraphen, den die beteiligten Knoten miteinander bilden (vgl. Abschnitt 3.1.2)
- die Mindestgröße jeder Kante (vgl. Abschnitt 3.1.7)
- die Konfiguration für jeden Knoten (vgl. Abschnitt 3.1.10)
- die Anzahl der Runden und pro Runde die Einstellungen für alle Knoten (vgl. Abschnitt 3.1.5)
- die Anzahl der Phasen pro Runde (vgl. Abschnitt 3.1.7)
- pro Phase die Menge der
  - aktiven Ports (vgl. Abschnitt 3.1.7)
  - Beobachter (vgl. Abschnitt 3.1.4)
  - Protokollierer (vgl. Abschnitt 3.1.4)
  - Knoten, die als neue Leser an einer Kante eine leere Kante sehen sollen (vgl. Abschnitt 3.1.7)

Auf welche Weise diese Informationen angeben werden müssen und wie diese ihren Weg zum Manager finden ist, wie eingangs in diesem Kapitel bereits erwähnt, Teil der Diplomarbeit von Stephanie Wist. Wichtig für diese Arbeit ist nur, dass der Manager der Simulation diese Informationen zukommen lässt.

3.2 Das verteilte System

Die folgenden Abschnitte geben eine Übersicht über die Teilnehmer des verteilten Systems, die für diese Arbeit relevant sind und im Folgenden als *ICE Objekte* bezeichnet werden. Zusätzlich erläutern sie deren Schnittstellen für die Kommunikation untereinander.

3.2.1 Manager

Der Manager ist die zentrale Einheit im verteilten System. Seine Aufgabe ist die Verwaltung der Simulationen. Er sichert außerdem Zwischenzustände und Simulationsergebnisse und ist der Ansprechpartner für die Anwenderprogramme des Benutzers. Er ist der einzige, der mit den Generatoren (vgl. Abschnitt 3.2.3) und den Simulationsobjekten (vgl. Abschnitt 3.2.2) kommuniziert. Behandelt wird der Manager in [41].

3.2.2 Simulationsobjekte

Eine laufende Simulation ist ein Zusammenspiel einer Laufzeitumgebung und beliebigem von Entwicklern stammenden Code. Es stellt sich daher die Frage, wie dieses Zusammenspiel am besten organisiert wird.

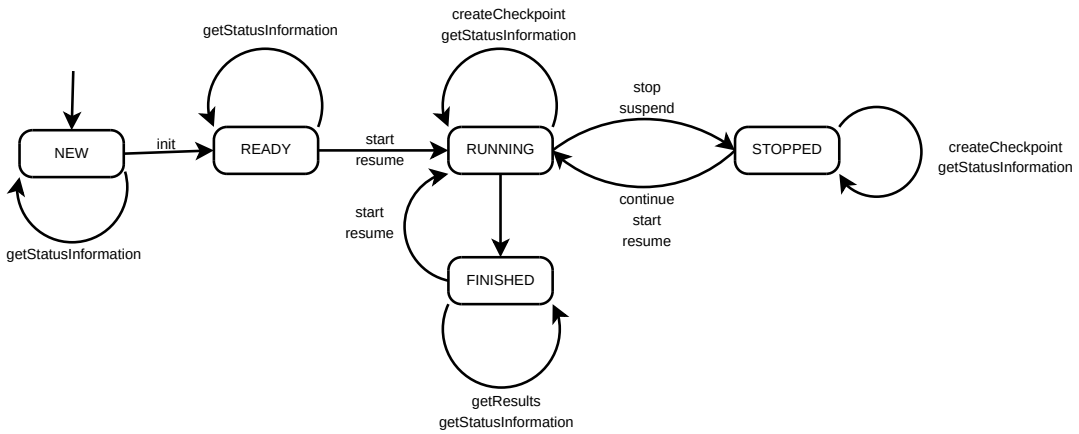


Abbildung 3.6: Zustandsautomat eines Simulationsobjektes

Eine naheliegende Möglichkeit ist es, einen zentralen Prozess mit einer Laufzeitumgebung zu haben. Zu jeder eingereichten Simulationsbeschreibung erzeugt die Laufzeitumgebung eine entsprechende Struktur und startet die Algorithmen. Dieser Ansatz bringt allerdings zwei große Nachteile mit sich.

Erstens bedeutet die Tatsache, dass in den Simulationen ein vom Benutzer stammender Code ausgeführt wird, dass im Wirtsprozess von einer negativen *Assertion* (vgl. Ansi C bzw. ISO/IEC 9899:1990) über einem Speicherzugriffsfehler bis hin zu einer Fließkommaausnahme alles auftreten kann. Fehlerhafter Code eines Benutzers führt also zum Abbruch aller im selben Prozess gerechneten Simulationen.

Zweitens ergibt sich für den zentralen Prozess ein Problem, wenn in einer neuen Simulation bisher nicht verwendete Algorithmen simuliert werden sollen. Den dazugehörigen Code muss die Laufzeitumgebung in diesem Fall nämlich dynamisch nachladen. Wie Abschnitt 4.1 diskutiert, ist dieser Teil von Lethe allerdings in C++ geschrieben und in dieser Sprache existiert dafür keine portable Möglichkeit. Deshalb müsste eine plattformspezifische Abstraktionsschicht eingefügt werden. Das bedeutet einen extra Aufwand und erschwert die Portierung der Software.

Verwendet man statt dessen pro Simulation einen eigenen Prozess, ergeben sich diese Nachteile nicht. Erstens ist diese Lösung robust gegen Fehler im Benutzercode, da die einzelnen Prozesse voneinander unabhängig sind. Zweitens muss kein Code dynamisch nachgeladen werden, weil von vorneherein der gebrauchte Code zu einer ausführbaren Datei zusammengelinkt werden kann. Diese Aufgabe erledigt der Generator (vgl. Abschnitt 3.2.3). Zu guter Letzt ist ein Vorteil dieser Lösung, dass das Scheduling zwischen den Simulationen vom Betriebssystem erledigt wird.

Ein solcher Prozess beherbergt genau ein *Simulationsobjekt*. Dieses wird ausschließlich vom Manager gesteuert. Die Schnittstelle bietet alle für Lethe essentiellen Funktionalitäten und wird als Zustandsautomat beschrieben (vgl. Abbildung 3.6).

Eine Simulation wird genau einmal initialisiert (`init`). Dabei gibt ihr der Manager die Simulationsbeschreibung. Daraufhin erstellt die Laufzeitumgebung der Simulation

alle nötigen Strukturen und wartet auf weitere Anweisungen. Der Manager kann nun entweder eine Runde der Simulation starten (`start`) oder einen gespeicherten Zwischenzustand einer Runde laden und fortsetzen lassen (`resume`).

Der Manager kann eine laufende Simulation jederzeit anhalten (`stop`). Optional kann er dabei den aktuellen Zwischenzustand anfordern (`suspend`), den er später ggf. auf einem anderen Computer fortsetzen lassen kann.

Hat der Manager eine Simulation mit oder ohne Erhebung des Zwischenzustandes angehalten, existieren für ihn drei Möglichkeiten. Erstens kann er die Simulation einfach weiterlaufen lassen (`continue`). Zweitens kann er einen Zwischenzustand laden und diesen fortsetzen lassen (`resume`). Drittens kann er einfach eine beliebige Runde der Simulation starten lassen (`start`). In den letzten beiden Fällen wird der aktuelle Zwischenzustand der Simulation verworfen und ist, falls der Manager sich ihn nicht zuvor hat geben lassen, verloren.

Unabhängig davon, ob die Simulation momentan läuft oder angehalten wurde, kann sich der Manager den aktuellen Zwischenzustand geben lassen (`createCheckpoint`). Ist die Simulation im Zustand `RUNNING` löst das Kommando intern einen `suspend` gefolgt von einem `continue` aus.

Wie Abschnitt 3.1.9 erläutert, kann beliebig viel Zeit vergehen, bis ein Zwischenzustand erhoben werden kann. Das liegt daran, dass die Simulation nur zu bestimmten Zeitpunkten überhaupt einen persistierbaren Zustand erreicht. Das bedeutet, dass es für das Kommando `suspend` und, falls die Simulation im Zustand `RUNNING` ist, auch für das Kommando `createCheckpoint` keine Garantie auf die Antwortzeit gibt.

Beim Übergang vom Zustand `RUNNING` in den Zustand `STOPPED` wird nicht unterschieden, ob ein Zwischenzustand erhoben wird oder nicht. Dadurch bleibt der Zustandsautomat schlicht. Allerdings bedeutet das, dass es für das Kommando `stop` ebenfalls keine Garantie auf die Antwortzeit gibt, weil wie im Fall von `suspend` gewartet werden muss, bis die Simulation einen persistierbaren Zustand erreicht hat. Dafür gibt es allerdings die Garantie, dass das Kommando `createCheckpoint` sofort beantwortet wird, wenn sich die Simulation im Zustand `STOPPED` befindet.

Ist die Simulation der aktuellen Runde zu Ende, geht das Simulationsobjekt von alleine in den Zustand `FINISHED` und informiert den Manager. Der kann sich dann die Simulationsergebnisse, bei Bedarf auch wiederholt, geben lassen (`getResults`). Er kann außerdem eine neue Runde anfangen (`start`) oder einen Zwischenzustand laden und fortsetzen (`resume`) lassen.

Der Manager kann sich zu jedem Zeitpunkt Informationen über den aktuellen Status einer Simulation geben lassen (`getStatusInformation`). Dazu gehören der Zustand, in dem sich die Simulation befindet sowie Angaben zur Laufzeit und der Fortschritt. Im Falle eines Fehlers werden außerdem nähere Informationen zu dessen Ursache geliefert.

Schließlich kann der Manager den Simulationsprozess zu jedem beliebigen Zeitpunkt beenden lassen (`die`). Um die Abbildung 3.6 übersichtlich zu halten, sind die dazu gehörenden Zustandsübergänge und der Endzustand selber nicht eingezeichnet.

Wie bereits erwähnt, bietet die Schnittstelle des Managers die Möglichkeit, ihn zu informieren, wenn die Simulation das Ende einer Runde erreicht hat. Zusätzlich bietet sie die Möglichkeit, den Manager zu benachrichtigen, wenn während der Simulation ein

Fehler auftritt. Wie Abschnitt 3.2.5 allerdings diskutiert, übernimmt der Proxy dieses Aufgabe.

3.2.3 Der Generator

Wie Abschnitt 3.2.2 erläutert, muss für jede Simulation eine eigene ausführbare Datei übersetzt und gestartet werden. Diese Aufgabe übernimmt der *Generator*.

Auf jedem teilnehmenden Computer läuft eine Instanz eines Generators. Hat der Manager entschieden, auf welchem Computer eine Simulation ausgeführt werden soll, kontaktiert er den entsprechenden Generator. Sobald der Simulationsprozess läuft, gibt der Generator dem Manager die Referenz auf das Simulationsobjekt, so dass dieser in Zukunft direkt mit der Simulation kommunizieren kann.

Die Übersetzung des Quellcodes zur ausführbaren Daten wird durch das GNU make [25] Programm gemacht. Dafür existieren eine Reihe von Skript- und Konfigurationsdateien, die in ihrer Gesamtheit das Buildsystem für Simulationen ausmachen. Der Generator kann über dieses Buildsystem in einem ersten Schritt den Quellcode übersetzen und in einem zweiten Schritt die daraus entstanden Bibliotheken zusammenlinken lassen.

Schlägt einer der beiden Schritte fehl, wird der Benutzer darüber informiert. Leider bekommt er dabei die Ausgaben des Compilers oder Linkers zu sehen und muss daraus erkennen, wo der Fehler liegt. Abschnitt 6.4.2 diskutiert, wie der Benutzer dabei unterstützt werden kann.

Damit nur der benötigte Benutzercode übersetzt und mit in das Simulationsprogramm gelinkt wird, muss der Generator am Anfang eine Konfigurationsdatei für das Buildsystem erzeugen, in der die Abhängigkeiten in geeigneter Form enthalten sind.

Wie Abschnitt 3.1.3 diskutiert, muss der Hypergraph aus der Simulationsbeschreibung im Speicher aufgebaut werden, damit das Scheduling funktioniert. Ein entsprechender Algorithmus der Laufzeitumgebung einer Simulation muss generisch sein, um jede mögliche Simulationsbeschreibung verarbeiten zu können. Das ist komplex und bringt dabei keinen Vorteil gegenüber der folgenden Lösung:

Der Manager gibt dem Generator auch die Simulationsbeschreibung. Damit stehen diesem alle nötigen Informationen zur Verfügung, um einen spezialisierten Code zu generieren, der später in der laufenden Simulation den Graphen aufbaut. Der generierte Code wird übersetzt und als Bibliothek mit in das Simulationsprogramm gelinkt.

Die Schnittstelle, über die der Manager mit dem Generator kommuniziert, ist extrem einfach. In einem einzigen Aufruf gibt der Manager dem Generator eine Simulationsbeschreibung und erhält nach einiger Zeit die Referenz auf das neu entstandene Simulationsobjekt oder eine Fehlermeldung, falls beim Compilieren, Linken oder Starten ein Fehler aufgetreten ist.

3.2.4 Source-Service

Das Konzept der Source-Services löst das Problem der Verteilung des Quellcodes. Es ermöglicht die garantierte Verwendung von verschiedenen Softwareversionen im verteil-

ten System. Außerdem erlaubt es einem Benutzer, seinen verwendeten Code geheim zu halten.

Behandelt wird dieses Konzept in der Diplomarbeit von Stephanie Wist. Wichtig für diese Arbeit ist, dass der Generator den Quellcode, der zur Ausführung kommen soll, von den Source-Services erhält.

Für die Übertragung des Quellcodes wird der IcePatch2 Dienst verwendet (vgl. ICE Benutzerhandbuch, Kapitel 43 [21]). Die Informationen über den Ort der Source-Services sind in der Simulationsbeschreibung enthalten. Außerdem besitzt jede Simulationsbeschreibung eine eindeutige Identifikation, die vom Manager gesetzt wird. Mit Hilfe dieser Identifikation wird garantiert, dass der Generator die richtige Version des Quellcodes vom Source-Service erhält.

Der Generator muss sich also bei allen in der Beschreibung aufgelisteten Source-Services unter Nennung der Identifikation melden und kann anschließend entsprechend viele IcePatch Clients starten. Am Ende dieses Vorgangs ist der richtige Quellcode lokal vorhanden und kann wie im Abschnitt 3.2.3 beschrieben gebaut werden.

3.2.5 Proxies

Die Verwendung von Benutzercode in den Simulationen kann, wie schon erwähnt, dazu führen, dass eine Simulation plötzlich abgebrochen wird. Für die Fehlersuche ist es sehr hilfreich, wenn der Benutzer oder der Entwickler in so einem Fall den Grund für den Abbruch erfahren kann.

Zu diesem Zweck wurden die Proxies eingeführt. Das ist pro Simulation ein ICE *Servant* (vgl. ICE Benutzerhandbuch Abschnitt 2.2.2 [21]), der im Prozess des Generators ausgeführt wird. So lange eine Simulation läuft reicht er einfach ICE Aufrufe zwischen den Simulationen und dem Manager durch. Zwischen dem Proxy und dem Simulationsprozess besteht außer der ICE Verbindung eine Pipe, die die Standardausgabe und die Standardfehlerausgabe des Simulationsprozesses mit dem Proxy verbindet.

Beendet sich nun der Simulationsprozess, wird die Pipe geschlossen, worüber der Proxy informiert wird. Den Grund für das Beenden kann sich der Proxy anschließend vom Betriebssytem geben lassen, da der Generatorprozess der Vater des Simulationsprozesses war. Außerdem sind etwaige Fehlermeldungen aus dem Simulationsprozess in der Pipe. Der Proxy kann damit den Manager zuverlässig und ausführlich über aufgetretene Fehler informieren.

Zu guter Letzt besteht mit den Proxies eine einfache Möglichkeit, einen Simulationsprozess garantiert zu beenden, auch wenn dieser eventuell nicht mehr auf ICE Aufrufe reagiert.

Die Proxies sind also nahezu transparent. Deshalb besitzt ein Proxy zum einen die Schnittstelle des Simulationsobjektes und zum anderen die Schnittstelle des Managers. Hinzu kommt noch eine Methode, die das Bootstrapping des Simulationsobjektes ermöglicht.

Da zwischen dem Proxy und dem Simulationsobjekt eine ICE Verbindung besteht, müssen beide Objekte ICE Referenzen auf den Partner besitzen. Dazu gibt der Proxy seine Referenz per Kommandozeilenparameter an das Simulationsobjekt. Ist die Simu-

lation erfolgreich initialisiert, meldet sich das Simulationsobjekt über die Schnittstelle des Proxies und gibt ihm auf diesem Weg seine eigene Referenz.

Eine alternative Möglichkeit ist, dass sich das Simulationsobjekt beim Namensdienst (vgl. ICE Benutzerhandbuch, Kapitel 36 [21]) anmeldet und der Proxy den Namensdienst so lange fragt, bis der entsprechende Eintrag vorhanden ist. Allerdings muss der Proxy dafür den Namen des Simulationsobjektes kennen. Wenn er diesen per Kommandozeilenparameter festlegen würde, wäre gegenüber der obigen Lösung nichts gewonnen. Außerdem ist die Transparenz der Proxies beeinträchtigt, wenn sowohl der Proxy als auch das Simulationsobjekt im Namensdienst eingetragen sind. Zugegebenermaßen ist Letzteres eine ästhetische Entscheidung.

3.2.6 Der Breeze-Server

Während einer Simulation können vielerlei Dinge geschehen, über die der Benutzer gerne informiert ist. Zu diesem Zweck bietet Lethe eine an Syslog [40] angelehnte Schnittstelle zum Ausgeben von Lognachrichten. Diese Nachrichten werden über die ICE Middleware an einen zentralen Logging Server, den sogenannten *Breeze-Server*, geschickt.

Die ursprüngliche Idee für Lethe war, den *IceStorm* Dienst zu verwenden (vgl. ICE Benutzerhandbuch, Kap. 42 [21]). Dieses Publisher-Subscriber System würde es ermöglichen, alle Interessierten sofort über ein Ereignis zu benachrichtigen. Allerdings persistiert IceStorm nicht. Der Benutzer möchte aber z.B. am nächsten Morgen wissen, was über Nacht mit seinen Simulationen geschehen ist. Man muss also den IceStorm Dienst dahingehend erweitern, dass er Ereignisse persistiert und auf Anfrage wieder versendet.

Aus Zeitgründen wurde diese Idee allerdings gegen eine einfachere Lösung, den Breeze-Server, ersetzt. Dieser Dienst persistiert alle Nachrichten, kann diese aber nicht pushen. Statt dessen muss jeder Interessierte den Dienst regelmäßig abfragen, um auf dem Laufenden zu bleiben.

Eine Nachricht besteht dabei aus einem Typ, einer Herkunft, einem Text und einem Zeitstempel, der vom Breeze-Server bei Eingang der Nachricht gesetzt wird. Auf alle diese Felder kann serverseitig gefiltert werden, so dass man sich z.B. nur die kritischen Fehler einer bestimmten Simulation seit gestern Abend anzeigen lassen kann.

Im Folgenden sind die Typen einer Lognachricht und ihr gedachter Einsatzweck definiert.

emergency das System ist unbenutzbar

alert ein sofortiger Benutzereingriff ist erforderlich

critical ein kritisches Ereignis ist eingetreten

error ein Fehlerereignis ist eingetreten

warning es ist ein Ereignis eingetreten, das Probleme machen könnte

notice es ist ein normales aber dennoch wichtiges Ereignis eingetreten

info es ist ein normales Ereignis eingetreten

debug es ist ein Ereignis eingetreten, das zum Zweck der Fehlersuche festgehalten wird.

Zusätzlich zum Breeze-Server gehören *publish* und *lastlog*, zwei Konsolenprogramme zum Versenden und Abfragen von Lognachrichten, zum Umfang von Lethe. Vor allem Letzteres kann vom Benutzer direkt eingesetzt werden, um über Ereignisse in den Simulation auf dem Laufenden zu bleiben. Der Anhang B.2 beschreibt diese beiden Programme und die Konfigurationsmöglichkeiten des Breeze-Servers.

4 Implementierung

Dieses Kapitel beschäftigt sich mit der Umsetzung der im Entwurf gefundenen Lösungen. Zuerst klärt Abschnitt 4.1, in welcher Programmiersprache Lethe implementiert ist.

Anschließend erörtert Abschnitt 4.2 die Implementierung der Kanten, die sogenannten *Streams*. Dort stellen sich die Fragen nach der Größe des Puffers und der Persistierung des Zustandes. Da Streams die von den Knoten verwendeten Daten transportieren müssen, stellt sich außerdem die Frage, wie man die Verwendung von beliebigen numerischen Bibliotheken ermöglichen kann.

Bei der Diskussion dieser Fragen kommen weitere Probleme wie z.B. die Konsistenzsicherung der Streams auf. Abschnitt 4.3 erklärt wie diese Probleme durch die *Ports*, die Verbindungsstücke zwischen Knoten und Kanten, gelöst werden.

Abschnitt 4.4 beschäftigt sich mit der Implementierung der Knoten, den sogenannten *Modulen*. Insbesondere erklärt dieser Abschnitt die Programmierschnittstelle (API) von Lethe.

Sowohl die Streams als auch die Module sind in zwei Schichten aufgeteilt. Die Abschnitte 4.2 und 4.4 behandeln jeweils nur die untere Schicht, die für den Datentransport und die Datenmanipulation zuständig ist. Die obere Schicht ist verantwortlich für das Scheduling. Abschnitt 4.5 beschäftigt sich mit dieser Schicht und erläutert, wie der Algorithmus aus Abschnitt 3.1.3 umgesetzt wird.

Die Implementierung der Teile von Lethe, die für das Verständnis der API und der Laufzeitumgebung irrelevant sind und keine Besonderheiten aufweisen, werden in diesem Kapitel nicht behandelt. Der Grund dafür ist schlichtweg, dass eine komplette Softwarebeschreibung den Rahmen dieser Arbeit bei weitem sprengen würde. Die betroffenen Teile sind der Generator (vgl. Abschnitt 3.2.3), der Proxy (vgl. Abschnitt 3.2.5), der Breeze Server (vgl. Abschnitt 3.2.6) und Details der Laufzeitumgebung. Auch das Buildsystem (vgl. Abschnitt 3.2.3) wird in dieser Arbeit nicht näher beschrieben. Nur die Teile, die den Entwickler direkt betreffen, werden erläutert. Der Interessierte wird auf den Quellcode verwiesen, der durch aussagekräftige Bezeichner und geeignete Kommentare leicht verständlich ist. Er ist dieser Arbeit auf einer CD beigefügt.

4.1 Die Wahl der Programmiersprache

Der Entwickler sollte die Algorithmen in derselben Sprache implementieren, in der die Laufzeitumgebung geschrieben ist. Alles andere kostet auf Grund von Zwischenschichten unnötig Laufzeit. Da ein Anwender im Normalfall nicht bereit ist, eine neue Sprache zu lernen, um eine Software einzusetzen, muss bei der Wahl dieser Sprache berücksichtigt werden, welche Kenntnisse typischerweise beim Anwender vorhanden sind.

Die Erfahrung zeigt, dass bei Kanalcodierern hauptsächlich Kenntnisse über Matlab und C vorhanden sind. Matlab ist, wie Abschnitt 2.1 erläutert, eine auf numerische

Berechnungen spezialisierte Sprache. Sie ist für Programme mit anderen Schwerpunkten nicht geeignet. Da die Laufzeitumgebung nahezu keine numerischen Berechnungen durchführt, statt dessen aber die Simulation verwalten und eine Anbindung an das verteilte System bereitstellen muss, scheidet Matlab aus.

Allerdings ist C auch keine gute Wahl. Aus Mangel an wesentlichen Konzepten muss vieles von Hand implementiert werden, was zu einem umfangreichen und schwer verständlichen Code führt. Das war mitunter die Motivation für die Entwicklung vieler neuer Programmiersprachen. Eine dieser Sprachen ist C++. Wie Abschnitt 2.5 erklärt, ist C++ bis auf wenige Details eine Obermenge von C. Der Lernaufwand für den Entwickler kann also in Grenzen gehalten werden.

Vor allem die Unterstützung von objektorientiertem Programmieren war ausschlaggebend für die Entscheidung, das Framework in C++ zu implementieren. Die Programmierschnittstelle für Entwickler ist dabei so gestaltet, dass die Unterschiede zwischen C und C++ möglichst keine Rolle spielen.

Doch wie Abschnitt 2.5 auch erklärt, schneidet C++ im Vergleich mit moderneren Programmiersprachen schlecht ab. Der Grund dafür ist vor allem, dass C++ viele Altlasten von C geerbt hat. Lassen es die Anforderungen zu, eine langsamere Ausführung des Programmes zu erlauben, ist einem Programmierer deshalb gut geraten, wenn er eine modernere Sprache verwendet.

Der Generator, die Proxies und der Breeze Server sind in Python [32] geschrieben. Der Grund hierfür ist, dass Python durch elegante Konzepte wie die dynamische Typisierung sehr kurze Entwicklungszeiten und die Erstellung von kompaktem Code ermöglicht. Die betroffenen Teile sind der Generator, die Proxies und der Breeze Server. Da die Implementierungen dieser Teile, wie eingangs in diesem Kapitel erwähnt, in dieser Arbeit nicht behandelt werden, wird auf eine genauere Beschreibung von Python im Rahmen dieser Arbeit verzichtet. Der Interessierte wird auf die umfangreiche Dokumentation unter [32] verwiesen.

4.2 Streams

Ein Stream hat zwei Aufgaben. Erstens ist er wesentlich für das Scheduling verantwortlich und muss seine Leser und Schreiber implizit aufrufen. Zweitens muss er die Daten des Schreibers puffern und allen Lesern zur Verfügung stellen. Die erste Aufgabe übernimmt die Klasse `Stream`. Die zweite Aufgabe übernimmt die davon abgeleitete Klasse `Templates::Stream`[1]. Beide Klassen sind im Namensraum `::Core` definiert.

Diese Aufteilung der Aufgaben in zwei Schichten dient der Trennung von Datenfluss und Schedulinglogik. In diesem Abschnitt wird nur die Implementierung der `Templates::Stream` Klasse behandelt. Die Aufgaben der `Stream`-Klasse erörtert ein eigener Abschnitt über das Scheduling (Abschnit 4.5).

Da die Streams die zentrale Infrastruktur der Simulationen sind, ist vor allem die Geschwindigkeit zur Laufzeit maßgeblich für Entscheidungen über die Implementierung. Die ersten Konsequenzen daraus ergeben sich in Abschnitt 4.2.1. Er erklärt, wie der

[1] Zur Notation von Namensräumen siehe Abschnitt 2.5

Puffer in einem Stream implementiert ist. Abschnitt 4.2.2 leitet anschließend her, wie groß ein Puffer mindestens sein muss und welche Größe Lethe für die Puffer wählt.

Abschnitt 4.2.3 führt die sogenannten *Basistypen* ein. Diese ermöglichen es, dass der Zustand eines Streams erfasst werden kann und stellen außerdem sicher, dass alle durch den Stream verbundenen Module miteinander kommunizieren können. Insbesondere erläutert dieser Abschnitt, dass `Templates::Stream` eine Template-Klasse ist und diskutiert, warum das so sein muss.

Um die Verwendung beliebiger numerischer Bibliotheken zu ermöglichen führt Abschnitt 4.2.4 die sogenannten *Arbeitstypen* ein. Mit Hilfe von Konvertierungsfunktionen zwischen Arbeitstypen und Basistypen wird es möglich, dass ein Modul einen anderen Typ für die Daten verwendet als der Stream. Insbesondere ist es dadurch möglich, dass zwei durch einen Stream verbundene Module unterschiedliche Bibliotheken verwenden.

Abschnitt 4.2.5 diskutiert anschließend eine Optimierung, die unnötiges Konvertieren verhindert, um im Normalfall trotz Arbeitstypen eine hohe Ausführungsgeschwindigkeit zu erhalten.

4.2.1 Der Puffer

Ein Stream ist eine “first-in-first-out“ Warteschlange (FIFO). Die geschriebenen Daten müssen unverändert und in der Schreibreihenfolge den Lesern zur Verfügung gestellt werden. Wie Abschnitt 3.1.2 erläutert, muss dabei garantiert werden, dass alle Leser dieselben Daten sehen.

Für die Implementierung der Streams sind zwei Dinge entscheidend, um hohe Geschwindigkeiten zur Laufzeit zu erreichen: erstens die Verwendung von statischem Speicher und zweitens das Prinzip, keine Daten zu kopieren.

Die Konsequenz des ersten Punktes ist, dass sämtliche dynamische Datenstrukturen wie z.B. Queues ausscheiden. Statt dessen werden Ringpuffer eingesetzt. Diese Datenstruktur ist altbekannt und hat sich für den Einsatz in FIFOs bewährt. Im Gegensatz zu einem typischen Ringpuffer braucht der Ringpuffer eines Streams mehrere Lesezeiger, die unabhängig voneinander bei Leseaufrufen der entsprechenden Module erhöht werden. Ein Puffer ist deshalb genau dann voll, wenn der Schreibzeiger einen beliebigen Lesezeiger erreicht hat. Ansonsten ist die Implementierung die eines typischen Ringpuffers, so dass hier nicht näher darauf eingegangen wird.

Die Konsequenz des zweiten Punktes ist, dass die Leser und Schreiber direkte Zeiger in den Puffer erhalten. Dadurch kann ein Modul die Daten zwischen zwei Streams abbilden, ohne dass dazwischen kopiert werden muss. Allerdings ergeben sich dadurch zwei Probleme. Erstens ist der Umbruch am Ende des Puffers ein Sonderfall, der behandelt werden muss. Zweitens vergeht beliebig viel Zeit zwischen dem Schreibzugriff eines Moduls und der tatsächlichen Aktualisierung der Daten im Puffer. Dadurch können zwischenzeitlich durch implizites Aufrufen andere Module auf den Stream zugreifen. Dasselbe gilt analog für Lesezugriffe. Die Konsistenz der Streams muss also gesichert werden. Abschnitt 4.3.1 erklärt, wie die Ports die beiden obigen Probleme lösen.

4.2.2 Die Größe eines Puffers

Falls der Lese- bzw. Schreibaufruf eines Moduls zu einem impliziten Aufruf des Schreibers bzw. des Lesers führt, darf es nicht passieren, dass der anschließende Schreib- bzw. Leseaufruf auf demselben Stream wiederum zu einem impliziten Aufruf des ursprünglichen Moduls führt. Denn dann liegt eine zirkuläre Warteabhängigkeit vor. Neben der Tatsache, dass das Scheduling so etwas verbietet (vgl. 3.1.6), ist das Problem an dieser Situation, dass sie eine Verklemmung zwischen den beiden Modulen darstellt, gegen die weder der Benutzer noch der Entwickler etwas unternehmen kann. Formal lässt sich das folgendermaßen beschreiben:

Gegeben sei ein Puffer, der maximal b Einheiten aufnehmen kann, ein Leser, der in einem Aufruf r Einheiten liest und ein Schreiber, der in einem Aufruf w Einheiten schreibt. Wenn der Puffer mit a Einheiten gefüllt ist, dann ist die notwendige Bedingung für eine zirkuläre Warteabhängigkeit, unabhängig davon, ob der Leser oder der Schreiber als erster beginnt, folgendermaßen gegeben:

$$a < r \ \wedge \ b - a < w \tag{4.1}$$

Um verhindern zu können, dass diese Bedingung jemals zutrifft, braucht Lethe vom Entwickler die Information, wie viele Einheiten pro Aufruf maximal gelesen bzw. geschrieben werden. Diese Anzahl wird die Blockgröße genannt und ist eine Eigenschaft eines Ports.

Angenommen, der obige Stream besitzt m Leser mit den Blockgrößen $r_i, \ i \in \{1, \ldots, m\}$ und n Schreiber mit den Blockgrößen $w_j, \ j \in \{1, \ldots, n\}$. Dann existiert kein a, das für beliebige i und j die Bedingung 4.1 erfüllt, genau dann, wenn

$$b \geq \max_{i \in \{1,\ldots,m\}} r_i - 1 + \max_{j \in \{1,\ldots,n\}} w_j \tag{4.2}$$

Letztendlich heißt das, wenn der Füllstand für die größte Leser-Blockgröße gerade um eins zu klein ist, dann muss im Puffer mindestens so viel Platz sein, dass die größte Schreiber-Blockgröße noch hinein passt.

Neben diesem Minimum muss Lethe noch die Mindestgröße berücksichtigen, die der Benutzer für einen Stream angeben kann (vgl. Abschnitt 3.1.7). Es ist nicht zu erwarten, dass eine Vergrößerung der Puffer über das Maximum dieser beiden Minima hinaus Vorteile mit sich bringt. Deshalb wählt Lethe für jede Puffergröße dieses Maximum.

4.2.3 Basistypen

Unterschiedliche Module können mit unterschiedlichen Datentypen arbeiten. Je nach Anwendungsfall kommen z.B. ganze oder komplexe Zahlen vor. Es ist auch denkbar, dass gemischte Tupel von beliebiger Größe verarbeitet werden. Im Allgemeinen lässt sich nicht abgrenzen, was als Repräsentationen der zwischen den Modulen ausgetauschten Daten in Frage kommt, und was nicht.

Deshalb bietet Lethe dem Entwickler die Möglichkeit, beliebige Typen zu spezifizieren. Die ausführliche Diskussion darüber findet in [41] statt. Wichtig für diese Arbeit

ist, dass alle von Entwicklern spezifizierten Typen als sogenannte *Basistypen* den Modulen und den Streams zur Verfügung stehen. Dazu generiert Lethe für jeden Basistyp ein Verzeichnis, in dem sich unter anderem eine Header-Datei befindet, die von beliebigen C++-Modulen eingebunden werden kann. Diese Datei definiert im Namensraum `::Extensions::Types` eine Klasse mit dem Namen des Basistyps. Auch der Name des Verzeichnisses und der Name der Header-Datei entspricht dem Namen des Basistyps. Eine weitere Datei in dem Verzeichnis des Basistyps ist die sogenannte `config.mak`, die zum Buildsystems gehört. Abschnitt 4.2.4 erklärt, welche Rolle sie spielt. Die Header-Datei für einen Basistyp wird im Folgenden als die Basistyp-Datei bezeichnet.

Für die Implementierung der Streams bedeutet die freie Wahl der Basistypen, dass der Puffer generisch sein muss, um Daten von beliebigem Typ verwalten zu können. Ein Stream ist also ein generischer Datencontainer. In C++ existiert für solche Zwecke das Konzept der Templates. Der Vorteil gegenüber alternativen Möglichkeiten wie z.B. die Verwendung einer abstrakten Oberklasse für Daten, ist die Effizienz zur Laufzeit. Alle nötigen Informationen über die verwendeten Typen sind zum Zeitpunkt der Übersetzung bekannt, so dass zur Laufzeit keine Entscheidungen mehr darüber getroffen werden müssen.

Die gepufferten Daten in einem Stream gehören zum Zustand des Streams. Damit dieser Zustand persistiert werden kann (vgl. Abschnit 3.1.9), muss es der ICE Laufzeitumgebung möglich sein, die Daten des jeweiligen Typs zu serialisieren. Deshalb generiert Lethe aus der Typspezifikation des Entwicklers zusätzlich eine Datei, die den Typ in der ICE Sprache für Schnittstellen spezifiziert (Slice, vgl. ICE Benutzerhandbuch, Kapitel 4 [21]). Der Slice-Compiler kann aus diesen Informationen die Funktionen zum Serialisieren und Deserialisieren generieren. Der Basistyp ist von diesem ICE-Typ abgeleitet und definiert die Funktionen `serialize` und `deserialize` und den Default-Konstruktor.

Die ersten beiden Funktionen werden gebraucht, da die ICE Funktionen zum Serialisieren und Deserialisieren eines Typs leider nicht im Namensraum dieses Typs existieren. Da die `Templates::Stream`-Klasse aber nur über den Typ ihres Puffers auf passende Funktionen zugreifen kann, muss der Basistyp welche definieren, die den Aufruf der jeweiligen ICE Funktionen kapseln. Der Grund für den Default-Konstruktor ist, dass der Entwickler, wie Abschnitt 4.2.4 erklärt, in der Klasse des Basistyps eventuell Konstruktoren zur Typkonvertierung definieren muss. Dadurch entfernt er die in C++ implizite Definition des Default-Konstruktors. Da dieser aber gebraucht wird, muss er explizit definiert werden.

Da es denkbar ist, dass ein Modul von zwei verschiedenen Streams Daten von unterschiedlichem Typ haben möchte, muss der Basistyp eine Eigenschaft eines Ports sein. Zwei Module können genau dann miteinander kommunizieren, wenn die durch den Stream verbundenen Ports denselben Basistyp verwenden. Anders formuliert bedeutet das, dass alle Ports, die an denselben Stream angeschlossen sind, denselben Basistyp verwenden müssen.

4.2.4 Arbeitstypen

Wie der vorherige Abschnitt erläutert, ist die besondere Eigenschaft eines Basistyps, dass die ICE Laufzeitumgebung diesen serialisieren kann. Das ist für beliebige andere Typen

nicht gegeben. Das bedeutet, dass insbesondere Typen aus numerischen Bibliotheken nicht als Basistyp verwendet werden können.

Damit der Entwickler trotzdem mit beliebigen numerischen Bibliotheken arbeiten kann, kennt Lethe das Konzept der *Arbeitstypen*. Das ist ein beliebiger Typ, der von einem Modul eingesetzt wird. Zu einem Arbeitstyp muss ein Basistyp existieren, der ihn inhaltlich aufnehmen kann. Definiert der Entwickler, wie ein Objekt des einen Typs in ein Objekt des anderen Typs und umgekehrt zu konvertieren ist, kann der Stream den Basistyp und das Modul den Arbeitstyp verwenden.

Um das zu verdeutlichen, sei angenommen, ein Modul muss komplexe Zahlen verarbeiten und der Entwickler möchte dafür die GSL (vgl. Abschnitt 2.6) verwenden. Das bedeutet, dass das Modul die komplexen Zahlen als Objekte des Types `gsl_complex` lesen und schreiben möchte. Der Basistyp des ein- und des ausgehenden Streams sei `Complex`, der die beiden Fließkommazahlen `real` und `imag` enthält. Um die Konvertierung zu ermöglichen muss der Entwickler in der Basistyp-Datei von `Complex` folgendes definieren:

```
Complex(const gsl_complex& c)
{
    real = GSL_REAL(c);
    imag = GSL_IMAG(c);
}

operator gsl_complex() const
{
    return gsl_complex_rect(real, imag);
}
```

Die erste Funktion ist der Konstruktor, der die beiden Variablen `real` und `imag` mit den entsprechenden Werten aus der übergebenen `gsl_complex` Zahl initialisiert. Dazu verwendet er die beiden GSL Macros `GSL_REAL` und `GSL_IMAG`, die den Real- bzw. Imaginärteil einer `gsl_complex` Zahl liefern. Mit Hilfe dieses Konstruktors kann überall ein Objekt vom Typ `gsl_complex` verwendet werden, wo ein Objekt vom Typ `Complex` erwartet wird.

Die zweite Funktion ist ein Castoperator. Er wird immer dann gerufen, wenn ein Objekt vom Typ `Complex` verwendet aber ein Objekt vom Typ `gsl_complex` erwartet wird. Mit Hilfe der GSL Funktion `gsl_complex_rect` erzeugt diese Funktion eine `gsl_complex` Zahl aus den Variablen `real` und `imag` und gibt sie zurück.

Der Arbeitstyp ist wie der Basistyp eine Eigenschaft eines Ports. Da es bei einfachen Modulen denkbar ist, dass gar keine numerische Bibliothek zum Einsatz kommt, entspricht der Arbeitstyp, falls vom Entwickler nicht anders angegeben, dem Basistyp.

Damit die GSL Funktionen und Typen bekannt sind, muss der Entwickler die passenden Header-Dateien in der Basistyp-Datei einbinden.

```
#include <gsl/gsl_complex_math.h>
```

Außerdem muss das Buildsystem darüber informiert werden, dass die GSL verwendet wird, damit die Bibliotheken beim Linken mit eingebunden werden. Dazu muss der Entwickler die folgenden Zeilen in die `config.mak` eintragen.

```
LDFLAGS += -lgsl -lgslcblas
CCFLAGS +=
```

Beide Zeilen sind make Kommandos [18]. Die erste fügt den Linkeroptionen die Anweisungen an, die von der GSL benötigten Bibliotheken mit zu verwenden. Mit der zweiten Zeile können spezielle Compileroptionen angegeben werden, die bei der Übersetzung des Basistyps mit verwendet werden. Allerdings ist das für das Beispiel nicht nötig.

4.2.5 Vermeidung von Konvertierungen

Die obigen Konvertierungen verbrauchen Laufzeit. Zum einen, weil die entsprechenden Berechnungen durchgeführt werden müssen und zum anderen, weil es sich dadurch nicht mehr vermeiden lässt, zu kopieren (vgl. Abschnitt 4.3.1). Da bei jeder Lese- und Schreiboperation eine Konvertierung durchgeführt werden muss und es während einer Simulation sehr viele solcher Operationen gibt, sind die Kosten in der Summe signifikant. Deshalb lohnt es sich, über eine Optimierung nachzudenken, die Konvertierungen vermeidet.

Verwenden alle Ports eines Streams denselben Arbeitstyp, ist es unnötig, dass beim Schreiben vom Arbeitstyp in den Basistyp hin und beim Lesen vom Basistyp in den Arbeitstyp zurück konvertiert wird. Es genügt, wenn die gepufferten Daten nur beim Erfassen des Zustandes in den Basistyp konvertiert werden, damit sie mit Hilfe der ICE Laufzeitumgebung persistiert werden können. Die Anzahl der Konvertierungen ist damit durch die Größe des Puffers beschränkt. Verglichen mit der Anzahl von Konvertierungen, die in der Summe durch die Lese- und Schreiboperationen zustande kommen, ist das eine erhebliche Reduzierung.

Um diese Tatsache auszunutzen, ist ein Stream neben dem Basistyp mit dem Puffertyp parametrisiert. Der Puffertyp ist dabei der Typ, den der Puffer des Streams verwalten können muss. Sein Wert hängt von den Arbeitstypen der Ports ab. Falls alle Ports denselben Arbeitstyp haben, ist der Puffertyp gleich dem Arbeitstyp, was der oben beschriebenen Optimierung entspricht.

Haben die Ports aber unterschiedliche Arbeitstypen, bleibt nichts anderes übrig, als den Puffertyp auf den Basistyp zu setzen und beim Lesen und Schreiben zu konvertieren. Es wäre noch denkbar, einen der Arbeitstypen als Puffertyp zu verwenden, um so nur beim Lesen oder nur beim Schreiben konvertieren zu müssen. Allerdings bedeutet das, dass das Kreuzprodukt aller verwendeten numerischer Bibliotheken an Konvertierungsfunktionen existieren muss. Das würde die Basistypen extrem unübersichtlich und die Einführung einer neuen Bibliothek sehr aufwendig machen. Deshalb wird diese Optimierung nicht unterstützt.

Die Unterstützung für den Einsatz verschiedener numerischer Bibliotheken innerhalb einer Simulation beschränkt sich also darauf, ihn zu ermöglichen. Empfohlen wird allerdings, durchgängig dieselbe Bibliothek zu verwenden. In diesem Fall können durch

die Optimierung sehr viele Konvertierungen eingespart werden. Wie viel das ausmachen kann, klärt Kapitel 5.

4.3 Ports

Ports sind die Verbindungsstücke zwischen Modulen und Streams. Sie bieten dem Entwickler eine Schnittstelle für die Ein- und Ausgabe der Module. Die Eigenschaften eines Ports sind:

Name Um mehrere Ports eines Moduls voneinander unterscheiden zu können, besitzt jeder Port einen Namen. Überall, wo der Anwender mit einem Port arbeiten muss, kann er dessen Namen verwenden (vgl. Abschnitt 4.4.2.4). Da der Name auf Grund von Namenskonventionen in verschieden Kontexten anders geschrieben wird, müssen sich die Namen von zwei Ports durch mehr als nur die Groß-/Kleinschreibung unterscheiden.

Blockgröße Die Blockgröße ermöglicht es Lethe, die benötigte Puffergröße des angeschlossenen Streams zu berechnen (vgl. Abschnitt 4.2.2).

Basistyp Der Basistyp stellt sicher, dass zwei Ports, die durch einen Stream miteinander verbunden sind, miteinander kommunizieren können. Außerdem ermöglicht er es einem Stream, seinen Zustand zu persistieren (vgl. Abschnitt 4.2.3).

Arbeitstyp Der Arbeitstyp erlaubt im Zusammenspiel mit Konvertierungsfunktionen des Basistyps die Benutzung von beliebigen numerischen Bibliotheken (vgl. Abschnitt 4.2.4).

Die zentralen Datenstrukturen für die Ein- und Ausgabe sind die `Vector`-Klassen. Abschnitt 4.3.1 erklärt, wie sie gleich eine ganze Reihe von Problemen aus Abschnitt 4.2 lösen. Abschließend erläutert Abschnitt 4.3.2 die Implementierung der Ports durch die `Adapter`-Klasse.

4.3.1 Die Vector-Klassen

Die Kernidee der `Vector`-Klassen ist die folgende: anstatt eines Zeigers in den Puffer erhält der Entwickler bei Lese- bzw. Scheibzugriffen von den Ports ein Objekt vom Typ `ReadVector` bzw. `WriteVector`. Da die `Vector`-Klassen den Indexoperator definieren, ist diese Tatsache transparent für den Entwickler. Beim Zugriff über den Indexoperator können nun allerdings die folgenden Aufgaben erledigt werden:

Erstens wird eine optionale Bereichsprüfung vorgenommen, um definiert reagieren zu können, wenn der Entwickler einen Fehler macht und zu große Indizes verwendet. Die Überprüfung findet als Assertion statt, so dass sie bei Bedarf vom Compiler wegoptimiert werden kann.

Zweitens kann bei einem Zugriff, der über das Ende des Puffers hinausgehen würden, der Umbruch an den Pufferanfang durchgeführt werden. Hierfür benötigt der Vektor Informationen über die Größe des Puffers. Diese sind zusammen mit dem gültigen Bereich und dem Zeiger in den Puffer Parameter des Kontruktors.

Drittens kann auf transparente Weise eine Typkonvertierung durchgeführt werden. Für den `ReadVector` ist das trivial. Der Indexoperator gibt ein Objekt des Arbeitstyps zurück, verwendet zur Rückgabe allerdings ein Objekt des Basistyps. Falls sich die beiden Typen unterscheiden, sorgt der Compiler dafür, dass der Castoperator des Basistyps gerufen wird (vgl. Abschnitt 4.2.4). Die Daten werden dadurch effektiv kopiert. Unterscheiden sich die beiden Typen nicht, erzeugt der Compiler Code, der das Objekt unverändert und ohne Kopieren durchreicht.

Der `WriteVector` muss einen etwas größeren Aufwand treiben. Das liegt daran, dass ein Entwickler erwartet, dass er den Indexoperator auch für Lesezugriffe verwenden kann. Dabei muss er den Wert erhalten, den er zuvor an den gleichen Index geschrieben hat. In Abschnitt 4.4.2.4 ist ein Beispiel für einen solchen Fall. Bei einem solchen Zugriff muss also das bereits in den Basistyp konvertierte Datum in den Arbeitstyp zurück konvertiert werden. Nach der Aktualisierung muss das Datum wieder zurück in den Basistyp konvertiert werden. Es ist schwer zu garantieren, dass es allen iterativen Zugriffsmustern für alle denkbare Datentypen inklusive der Konvertierungsfunktionen egal ist, ob sie direkt auf den Daten oder auf Schattenkopien arbeiten, die bei jedem Zugriff hin und zurück konvertiert werden. Abgesehen vom Aufwand ist das der Grund, weshalb ein `WriteVector` einen temporären Puffer des Arbeitstyps besitzt, auf dem das Modul arbeitet. Sind alle Schreibzugriffe erledigt, wird dieser Puffer einmal objektweise in den Basistyp konvertiert und in den Puffer des Streams kopiert.

Damit der Aufwand eines temporären Puffers eingespart wird, wenn nicht konvertiert werden muss, existiert eine Template-Spezialisierung für den Fall, dass Arbeitstyp und Basistyp identisch sind. Dieser `WriteVector` ist trivial und entspricht dem `ReadVector.`

Eine noch zu klärende Frage ist, wann der Zeitpunkt erreicht ist, ab dem die Lese- oder Schreibzugriffe mit Sicherheit erledigt ist. Neben dem obigen `WriteVector`, der seinen temporären Puffer kopieren muss, ist das für die Konsistenzsicherung der Streams wichtig. Wie Abschnitt 4.2.1 erläutert, und wie im Falle des obigen `WriteVectors` deutlich wird, kann zwischen dem Schreibaufruf und dem Zeitpunkt der tatsächlichen Aktualisierung der Daten im Stream beliebig viel Zeit vergehen. Es kann deshalb durch implizite Aufrufen dazu kommen, dass andere Module auf denselben Stream zugreifen und somit vermeintlich geschriebene aber noch nicht aktualisierte Daten lesen. Auch kann es passieren, dass andere Module Daten überschreiben, die der Leser noch gar nicht gelesen hat. Um die Konsistenz der Streams zu sichern, dürfen die Lese- und Schreibzeiger des Ringpuffers erst erhöht werden, wenn mit Sicherheit keine weiteren Schreib- oder Lesezugriffe mehr gemacht werden.

Durch die Verwendung der Vektor-Objekte ist dieser Zeitpunkt allerdings genau definiert. Wenn das Objekt zerstört wird, können anschließend keine Zugriffe mehr gemacht werden. Deshalb ist der Destruktor der `Vector`-Klassen genau der richtige Ort, um die Lese- und Schreibzeiger zu aktualisieren. Um sicherzustellen, dass keine Kopien der Vektor-Objekte mehr existieren, führen die Vektor-Objekte eine Referenzzählung durch. Alternativ müsste das Kopieren von Vektor-Objekten verboten werden, was für einen Entwickler eine störende Einschränkung sein kann.

Die verzögerte Aktualisierung der Schreib- und Lesezeiger bewirkt, dass die Module auf Speicherbereichen des Puffers arbeiten, ohne dass diese währendessen als benutzt markiert sind. Allerdings stellt das kein Problem dar. Der Speicherbereich zwischen dem

Schreibzeiger und dem nächsten Lesezeiger wird von niemand anderem verwendet, da es nur einen Schreiber in jeder Phase gibt und die Lesezeiger den Schreibzeiger nicht überholen können. Der Speicherbereich zwischen einem Lesezeiger und dem Schreibzeiger wird maximal lesend von anderen Modulen verwendet, da der Schreibzeiger keine Lesezeiger überholen kann. Da die Lesezugriffe der einzelnen Module unabhängig voneinander sind, ist auch dieser Fall unproblematisch.

Greift ein Modul während eines Aufrufs mehrmals auf einen Stream zu, so muss der Entwickler aufpassen. Existiert zum Zeitpunkt des zweiten Zugriffes das erste Vektor-Objekt noch, so enthält das zweite Vektor-Objekt exakt dieselben Daten. Wie Abschnitt 4.3.2 beschreibt, wird dem Entwickler allerdings empfohlen, während eines Aufrufs maximal einmal auf einen Stream zuzugreifen. Möchte er dennoch mehrmals zugreifen, muss er geeignete Gültigkeitsbereiche der Vektor-Objekte verwenden.

Es ist wichtig, dass Zugriffe auf Streams ausschließlich über Vektor-Objekte geschehen. Bei einem direkten Zugriff über den Zeiger eines Objektes aus dem Vektor riskiert der Entwickler Probleme beim Umbruch am Pufferende. Sollte es für eine Bibliothek nötig sein, einen Zeiger auf den Datenbereich zu haben, bleibt dem Entwickler nichts anderes übrig, als einen lokalen Puffer anzulegen und die Daten aus dem Vektor dort hinein zu kopieren.

Falls alle Leser eines Streams beendet haben, ist es manchmal nötig, dass der Schreiber ganz normal weiterarbeiten kann (vgl. Abschnitt 3.1.4). Deshalb kennt die `WriteVector`-Klasse einen Modus, in dem sie auf jeden Fall einen temporären Puffer zum Beschreiben anlegt und diesen am Ende einfach verwirft.

4.3.2 Die Adapter-Klasse

Lethe generiert für jedes Modul eine passende Klasse, die die Anbindung an die Streams realisiert. Diese Klassen heißen `Adapter` und sind im Namensraum des jeweiligen Modul definiert. Abschnitt 4.4 erklärt die Details.

Für jeden Port eines Moduls existiert in der `Adapter`-Klasse eine Funktion mit dem Namen des Ports. Der Aufruf einer solchen Funktion löst je nach Typ des Ports eine Lese- oder Schreiboperation auf dem angeschlossenen Stream aus. Der Rückgabetyp einer solchen Funktion trägt auch den Namen des Ports. Beides wird in der `Adapter`-Klasse zum Komfort des Entwicklers entsprechend definiert.

Jede Port-Funktion besitzt einen Parameter, der die Anzahl der zu lesenden bzw. zu schreibenden Einheiten angibt. Das zurückgelieferte Vektor-Objekt bietet dann den Zugriff auf die angegeben Anzahl von Daten an. Jede Port-Funktion überprüft mittels einer Assertion, ob die angegeben Blockgröße des Ports tatsächlich eingehalten wird. Dadurch ist es möglich, auf entsprechende Fehler des Entwicklers definiert zu reagieren. Wie Abschnitt 4.2.2 erklärt, ist die Blockgröße die maximale Anzahl der Daten, auf die ein Modul während eines Aufrufs über einen Port zugreift. Wird sie nicht eingehalten, kann es zu Verklemmungen kommen. Die obige Sicherheitsmaßnahme wird allerdings umgangen, wenn ein Modul während eines Aufrufs mehrere Zugriffe auf einen Stream macht. Deshalb wird dem Entwickler empfohlen, pro Aufruf nur einmal auf einen Stream zuzugreifen. Soll ein Modul mehrmals auf einen Stream zugreifen, so muss der Entwickler

selber sicherstellen, dass die Blockgröße nicht überschritten wird. Wie Abschnitt 4.3.1 erläutert, muss er dabei außerdem auf die Lebensdauer der Vektor-Objekte achten.

Für eingehende Ports existiert eine zweite Funktion mit einem weiteren Parameter, der angibt, um wieviel Einheiten der Lesezeiger erhöht werden soll. Damit ist eine *Vorausschau*[2] auf die Daten möglich. Viele Filter müssen z.B. bei jedem Aufruf den aktuellen Wert zusammen mit den n nächsten berücksichtigen. Würde es für solche Module keine Vorrausschau geben, müssten sie die Daten zwischenspeichern. Diesen Mehraufwand kann man sich mit der obigen zweiten Funktion einfach einsparen.

Wie Abschnitt 3.1.4 erläutert, kann es bei Lese- und Schreiboperationen zu Situationen kommen, auf die die Laufzeitumgebung mit einem impliziten Beenden eines Moduls reagiert, falls das Modul den Sonderfall nicht behandelt. Implementiert ist dieser Mechanismus mit C++-Ausnahmen, die von der Laufzeitumgebung bei Eintritt der Sonderfälle geworfen werden. Das Fangen einer solchen Ausnahme entspricht der Behandlung des Sonderfalles, denn wird die Behandlungsfunktion des Moduls (vgl. Abschnitt 4.4.2.4) auf Grund einer nicht gefangenen Ausnahme verlassen, führt das immer zum impliziten Beenden des Moduls.

Die den ersten beiden Regeln aus 3.1.4 entsprechenden Ausnahmen sind die `BufferEmpty` Ausnahme und die `ModuleHasNoReaders` Ausnahme, die beide im Namensraum `::Core::Exceptions` definiert sind. Die dritte Regel nennt eine Möglichkeit für das Modul, über den Sonderfall informiert zu werden. Dazu existiert für ausgehende Ports eine zweite Funktion, mit einem zusätzlichen, boolschen Parameter. Hat dieser einen positiven Wahrheitswert, wirft die Laufzeitumgebung eine `::Core::Exceptions::BufferFull` Ausnahme, falls die Bedingungen dafür erfüllt sind. Das Modul kann diese Ausnahme entweder fangen oder sich durch sie implizit beenden lassen.

Das einzige Objekt vom Typ der `Adapter`-Klasse steht einem Modul unter dem Namen `ports` zur Verfügung. Über dieses Objekt erledigt der Entwickler alle Ein- und Ausgaben für sein Modul. In Abschnitt 4.4.2.4 findet sich ein Beispiel für die Verwendung des Adapter-Objektes.

In der `Adapter`-Klasse ist auch die Enumeration `Caller` definiert. Die Namen aller Ports tauchen darin als Konstanten auf. Die Behandlungsfunktion des Moduls (vgl. Abschnitt 4.4.2.4) erhält einen Parameter dieses Typs. Falls das Modul auf Grund eines impliziten Aufrufs gerufen wurde, kann das Modul an Hand des Wertes dieses Parameters erkennen, von welchem Stream der implizite Aufruf kommt. Auf diese Weise ist eine Sonderbehandlung von Anfragen einzelner Streams möglich. So kann z.B. ein Decoder bei einer adaptiven Decodierung erst dann Informationen in den Feedback Stream schreiben, wenn der Encoder diese anfordert (vgl. Abschnitt 3.1.6). Falls das Modul als initialer Knoten ausgewählt und deshalb vom Scheduler selber gerufen wurde, ist der Wert des Parameters `LETHE_SCHEDULER`[3].

[2] Es besteht eine Analogie zwischen Decodern und Parsern. Deshalb wurde dieser Begriff gewählt, der bei LR(k) Parsern [17] die gleichartige Technik bezeichnet.

[3] Die Verwendbarkeit des `Caller`-Parameter ist fraglich, da er nur von nicht-stationären Algorithmen berücksichtigt werden kann.

4.4 Module

Ein Modul hat zwei Aufgaben. Erstens muss es das Scheduling unterstützen und zweitens muss es Möglichkeiten zur Interaktion mit der Laufzeitumgebung bereitstellen. Die erste Aufgabe übernimmt die Klasse `::Core::Module`. Für die zweite Aufgabe generiert Lethe aus den Metainformationen des Entwicklers pro Modul sechs Klassen. Diese sind alle in einem Namensraum definiert, der in `::Extensions::Modules` liegt und den Namen des Moduls trägt.

Eine der generierten Klassen ist die `Adapter`-Klasse, die alle Ports des Moduls enthält. Sie wurde bereits in Abschnitt 4.3.2 behandelt. Die in Abschnitt 4.3 aufgelisteten Metainformationen über die Ports spielen für die Generierung dieser Klasse eine Rolle.

Neben der `Adapter`-Klasse werden Klassen für die Konfiguration, die Einstellungen, den Zustand und die Ergebnisse eines Moduls generiert. Abschnitt 4.4.1 nennt die Namen dieser vier Verwaltungsklassen und erklärt, wie sie der Entwickler verwenden kann.

Die zentrale Klasse unter den sechs generierten ist die `Module`-Klasse. Sie enthält alle Funktionen, die die Laufzeitumgebung aufruft, um mit dem Modul zu interagieren. Die Implementierung dieser Funktionen ist das einzige, was der Entwickler erledigen muss, nachdem Lethe die Klassen generiert hat. Abschnitt 4.4.2 erklärt anhand eines Beispiels diese Funktionen, die zusammen mit den Ports die Entwickler API von Lethe bilden.

Die `Module`-Klasse ist eine Template-Klasse. Der Grund dafür ist, dass der Typ der ein- und ausgehenden Streams erst fest steht, wenn der Graph der Simulation bekannt ist (vgl. Abschnitt 4.2.5) und ein Modul Referenzen auf seine Streams braucht (vgl. Abschnitt 4.5.1). Für jeden ein- und ausgehenden Stream besitzt die Template-Klasse also einen Parameter für den Typ des Streams.

Die `Module`-Klasse aus dem `::Extensions` Namensraum ist von `::Core::Module` abgeleitet. Die Vererbungsbeziehung zwischen den beiden `Modul`-Klassen erzeugt dieselbe zweischichtige Architektur wie bei den Streams. Ein eigener Abschnitt über das Scheduling behandelt die obere Schicht (Abschnitt 4.5). Folglich behandelt dieser Abschnitt hier nur die Klassen aus `::Extensions::Module`.

Die generierten Klassen stehen in mehreren Dateien, die alle in einem Verzeichnis mit dem Namen des Moduls liegen. Details dazu finden sich in [41]. Wichtig für diese Arbeit sind nur zwei dieser Dateien: eine Header-Datei mit dem Namen des Moduls und eines Datei mit dem Namen `config.mak`. Die erste Datei definiert die `Modul`-Klasse. Wie Abschnitt 2.5 erklärt, ist die unzureichende Unterstützung der C++ Compiler für Templates der Grund dafür, dass die Implementierung des Moduls in einer Header-Datei stehen muss. Diese Datei wird im Folgenden als die Modul-Datei bezeichnet. Analog zu Abschnitt 4.2.4 kann der Entwickler mit der zweiten Datei das Verhalten des Buildsystems steuern.

4.4.1 Verwaltungsklassen

Im Kapitel über den Entwurf von Lethe wird mehrfach erwähnt, dass der Entwickler spezielle Typen für ein Modul spezifizieren kann. Damit kann er kontrollieren, wie die Konfiguration, die Einstellungen, der Zustand und die Ergebnisse eines Moduls aussehen. Lethe generiert aus diesen Spezifikationen C++-Klassen (vgl. [41]).

Die generierte `Module`-Klasse enthält von jedem Typ ein Objekt. Der Zweck dieser Objekte ist zum einen, alle gleichartigen Informationen zu sammeln. Zum anderen verwenden die vordefinierten Funktionen der `Module`-Klasse diese Objekte, so dass der Entwickler in vielen Fällen die Funktionen nicht verändern muss. Die Definition der Objekte sieht in einem Modul folgendermaßen aus:

```
Lethe::Configuration _config;
Lethe::Settings _settings;
Lethe::State _state;
Lethe::Result _result;
```

Jeder Typ ist ein C++-struct, dessen Felder genau die Typen und Namen haben, die der Entwickler spezifiziert hat. `Lethe` ist ein Namensraum, der im Namensraum des Moduls existiert und alle generierten Typen enthält. Dadurch muss der Entwickler keine Rücksicht bei der Wahl von Bezeichnern nehmen.

Zu jedem dieser Typen existiert ein ICE Pendant damit die Inhalte der Variablen über das verteilte Netz transportiert werden können. Es ist für jeden Typ möglich, dass er für ein Modul nicht benötigt wird und deshalb leer ist. Da ICE die Definition von leeren Typen verbietet, fügt Lethe in einem solchen Fall eine einizige Variable ein. Für den Entwickler spielt das allerdings keine Rolle.

4.4.2 Die Entwickler API

Die folgenden Abschnitte definieren die API Funktionen von Lethe. Sie erwähnen dabei nur kurz den Zweck der Funktionen. Abschnitt 3.1.10 erläutert sie ausführlicher und erklärt vor allem das Zusammenspiel der Funktionen mit der Laufzeitumgebung.

Zur besseren Verständlichkeit findet die Erklärung dieser Funktionen anhand eines Beispielmoduls statt. Das Beispiel ist ein Gausskanal, der komplexen Zahlen am Eingang eine gaussverteilte komplexe Zufallszahl hinzuaddiert und sie in den Ausgang schreibt. Abschnitt 4.4.2.1 listet die Metainformationen dieses Moduls auf. Die Implementierung des Gausskanals verwendet die GSL (vgl. Abschnitt 2.6).

Alle Funktionen sind von Lethe vordefiniert. Die Zeilen, die der Entwickler im Beispiel hinzugefügt hat, sind mit einem + markiert. Es ist in keinem Fall nötig gewesen, dass eine generierte Zeile entfernt werden musste. Die vordefinierte Implementierung wurde extra so gewählt, dass dies im Allgemeinen nie nötig sein dürfte.

4.4.2.1 Metainformationen

Im Folgenden sind alle Metainformationen aufgelistet, die der Entwickler für den Gausskanal angeben muss.

- **Name:** GaussChannel
- **Ports:**
  - erster Port
    - **Name:** Input

        - **Typ:** eingehend
        - **Basistyp:** Complex
        - **Arbeitstyp:** gsl_complex
        - **Blockgröße:** 1
    - zweiter Port
        - **Name:** Output
        - **Typ:** ausgehend
        - **Basistyp:** Complex
        - **Arbeitstyp:** gsl_complex
        - **Blockgröße:** 1

- **Konfiguration**
    - Eine Enumeration, die für das Beispiel nur die Konstante `TAUS` definiert.
    - Eine Variable namens `typeofRng` vom Typ der Enumeration.
- **Einstellungen**
    - Eine Variable namens `sigma` vom Typ `double`. Das ist die Varianz der Wahrscheinlichkeitsverteilung.
    - Eine Variable namens `seed` vom Typ `unsigned long`. Das ist der Anfangswert des Zufallsgenerators.
- **Zustand**
    - Ein Byte-Array namens `rngState`. Das ist der Zustand des Zufallsgenerators.
- **Ergebnis:** Der Gauss Kanal liefert keine Ergebnisse. Der Typ ist deshalb leer.

4.4.2.2 Vorbereitungen

Der Gausskanal verwendet zum einen die komplexen Zahlen und zum anderen die Zufallsgeneratoren der GSL. Deshalb müssen am Anfang der Modul-Datei die entsprechenden Header-Dateien eingebunden werden.

```
+ 1: #include <gsl/gsl_complex_math.h>
+ 2: #include <gsl/gsl_randist.h>
```

Die für die Verwendung der GSL nötigen Bibliotheken werden schon auf Grund der `config.mak` des Basistyps vom Buildsystem berücksichtigt (vgl. Abschnitt 4.2.4). Außerdem sind für das Beispielmodul keine speziellen Compileroptionen nötig. Die `config.mak` des Moduls sieht deshalb folgendermaßen aus:

```
LDFLAGS +=
CCFLAGS +=
```

4.4.2.3 Der Logger

Wie Abschnitt 3.2.6 erläutert, bringt Lethe einen zentralen Logging Service mit. Damit Entwickler diesen verwenden können, existiert im Namensraum `Core` die Klasse `Logger`, die für jeden Nachrichtentyp eine Funktion definiert.

Beim Aufruf dieser Funktionen kann eine gewöhnliche printf-Syntax verwendet werden. Alternativ kann ein C++ String übergeben werden. Damit im Log zu erkennen ist, aus welchem Modul die Nachricht stammt, definiert die `::Core::Modul`-Klasse die Funktion `getVertexName`. Sie liefert den Namen, den der Benutzer bei der Erstellung der Simulation dem Knoten gegeben hat.

Um z.B. für eine Fehlersuche zu registrieren, dass irgendein Ereignis geschehen ist, kann der Entwickler eine der beiden folgenden Zeilen verwenden:

```
Core::Logger::debug("%s: something happened", getVertexName());
Core::Logger::debug(getVertexName() + ": something happened");
```

Jeder Aufruf des Loggers löst einen ICE Aufruf im Netzwerk aus, da die Meldung zum Breeze Server transportiert werden muss. Das muss synchron gemacht werden, damit eine Fehlersuche mittels Lognachrichten möglich ist. Es ist also zu beachten, dass die übermäßige Verwendung des Loggers die Laufzeit steigert.

Deshalb existiert für Debugging-Zwecke das Macro `DEBUGOUT`. Es bildet auf die Funktion `::Core::Logger::debug` ab, wenn die Simulation mit Debugging compiliert wird. Wird die Simulation ohne Debugging compiliert, werden alle `DEBUGOUT`-Aufrufe wegoptimiert. Das dafür verwendete Macro ist dasselbe Macro, das Assertions deaktiviert. Eingeschaltet wird es durch die Compileroption -DNDEBUG.

Der Aufruf von oben sieht mit dem `DEBUGOUT` Macro folgendermaßen aus:

```
DEBUGOUT(getVertexName() + ": something happened");
```

4.4.2.4 user_trigger

Das ist die zentrale Funktion eines Moduls. Hier werden eingehende Daten auf ausgehende Daten abgebildet. Gerufen wird die Funktion solange bis das Modul beendet. Ein Aufruf kommt dabei entweder direkt vom Scheduler oder implizit von einem Stream (vgl. Abschnitt 3.1.3).

```
  1: Decision user_trigger(Lethe::Caller caller) {

+ 2:     InputVector input = ports.input(1);
+ 3:     OutputVector output = ports.output(1);

+ 4:     GSL_REAL(output[0]) = \
+ 5:           GSL_REAL(input[0]) + \
+ 6:           gsl_ran_gaussian(_rng, _settings.sigma);
+ 7:     GSL_IMAG(output[0]) = \
+ 8:           GSL_IMAG(input[0]) + \
+ 9:           gsl_ran_gaussian(_rng, _settings.sigma);
```

```
10:     return CONTINUE;
11: }
```

Zeile 1 zeigt die Signatur der `user_trigger` Funktion. Der Rückgabewert ist vom Typ `Decision`. Dies ist eine Enumeration, die die beiden Konstanten `CONTINUE` und `QUIT` definiert. Durch Rückgabe der Konstanten `QUIT` signalisiert ein Modul der Laufzeitumgebung, dass es für die aktuelle Phase beenden möchte (vgl. Abschnitt 3.1.4). Die generierte Implementierung gibt am Ende der Funktion immer `CONTINUE` zurück, was für alle Module im Normalfall das korrekte Verhalten ist. Der einizige Parameter der Funktion ist vom Typ `Caller`. Wie Abschnitt 4.3.2 erklärt, kann das Modul an diesem Parameter erkennen, warum es gerufen wurde.

Die Zeilen 2 und 3 zeigen, wie der Zugriff auf ein- und ausgehende Streams über das in Abschnitt 4.3.2 eingeführte `ports` Objekt geschieht. `InputVector` und `OutputVector` sind Aliasnamen für passende Instanzen der `Vector`-Klassen (vgl. Abschnitt 4.3.1). Es ist in diesen Zeilen nicht zu erkennen, ob ein Lese- oder ein Schreibzugriff stattfindet. Deshalb ist es empfehlenswert, aussagekräftige Namen für die Ports zu verwenden. Da die Blockgröße für beide Ports eins ist, ist zu den verwendeten Parametern der Port-Funktionen nicht viel zu sagen.

Die Zeilen 4 bis 9 zeigen die lesenden und schreibenden Zugriffe auf die Daten der Streams mit Hilfe der Vektor-Objekte. Die `gsl_ran_gaussian` Funktion aus der GSL gibt eine gaussverteilte Zufallszahl zurück. Dazu verwendet sie den Zufallsgenerator `_rng` (vgl. nächster Abschnitt) Die Varianz der Zufallsverteilung ist dabei eine Einstellung des Moduls und wird direkt aus der `_settings`-Variablen gelesen.

Die `_config`, `_results` und `_state`-Variablen werden in diesem Beispiel in der `user_trigger` nicht benötigt. Muss ein Modul allerdings auf seine Konfiguration zugreifen, Ergebnisse protokollieren oder auf seinem Zustand arbeiten, sollte das am Besten direkt mit diesen Variablen geschehen. Denn dann kann der Entwickler die folgenden Funktionen unverändert verwenden. Der Grund, warum das Beispiel die `_state`-Variable nicht verwendet ist, dass sich der gesamte Zustand des Modul im `_rng`-Objekt befindet.

Das Beispiel ist übrigends ein Fall, in dem der Indexoperator eines `WriteVectors` zum Lesen der Daten verwendet wird (vgl. Abschnitt 4.3.1). Bei den beiden Zugriffen auf den ausgehenden Stream werden jeweils nur Teile des Objekte verändert.

4.4.2.5 user_init

Die `user_init` Funktion wird für jedes Modul genau einmal gerufen. Dabei erhält das Modul seine Konfiguration.

```
  1: void user_init(const Lethe::Configuration& config) {
  2:     _config = config;
+ 3:     switch (config.typeofRng) {
+ 4:         case Config::GaussChannel::TAUS:
+ 5:              _rng = gsl_rng_alloc(gsl_rng_taus); break;
+ 6:         /* ... */
+ 7:     }
```

```
  8: }
```

Die Implementierung der `user_init` Funktion braucht normalerweise keine Eingriffe des Entwicklers. Der generierte Code handelt in den meisten Fällen richtig, indem er die Konfiguration, wie in Zeile 2 zu sehen ist, abspeichert. Dadurch kann in der `user_trigger` Funktion auf die Konfiguration des Moduls zugegriffen werden.

Müssen, wie im Beispiel, interne Strukturen aufgebaut werden, sollte dies in der `user_trigger` Funktion geschehen. Die `gsl_rng_alloc` Funktion aus der GSL erzeugt ein Zufallsgenerator-Objekt. Dabei wird durch die Konfiguration des Gausskanal-Moduls gesteuert, von welchem Typ der Generator sein soll. Das Beispiel verwendet einen *Tausworthe* Zufallsgenerator ([29]). Für das Beispiel spielt der Typ allerdings keine weitere Rolle. Er dient hier nur dem Zweck, die Verwendung einer Konfiguration zu demonstrieren.

Damit die Variable für das Zufallsgenerator-Objekt definiert ist, muss noch folgende Zeile zur Modul-Datei hinzugefügt werden:

```
 + 1: gsl_rng* _rng;
```

4.4.2.6 user_finish

Die `user_finish` Funktion dient dazu, eventuell belegte Ressourcen wieder frei zu geben. Es ist garantiert, dass nach dem Aufruf dieser Funktion keine andere user-Funktion mehr gerufen wird.

```
  1: void user_finish() {
+ 2:      gsl_rng_free(_rng);
  3: }
```

Die generierte Funktion ist leer. Für das Gausskanal-Modul muss der Entwickler das zuvor erzeugte Zufallsgenerator-Objekt wieder freigeben (Zeile 2).

4.4.2.7 user_reset

Die `user_reset Funktion` wird immer gerufen, wenn eine neue Runde beginnt. Das Modul erhält dabei seine Einstellungen.

```
  1: void user_reset(const Lethe::Settings& settings) {
  2:     _settings = settings;
+ 3:     gsl_rng_set(_rng, settings.seed);
  4: }
```

Die generierte Implementierung speichert die neuen Einstellungen ab (Zeile 2). Dadurch kann die `user_trigger` Funktion immer auf die aktuellen Einstellungen zugreifen. Der Gausskanal muss an dieser Stelle seinen Zufallsgenerator zurücksetzen (Zeile 3). Dazu erhält er vom Benutzer über die Einstellungen den neuen Anfangswert.

4.4.2.8 user_newPhase

Der einzige Zweck der `user_newPhase` Funktion ist, die Module über den Beginn einer neuen Runde zu informieren.

```
1: void user_newPhase(int phaseNumber, int numberofPhases) {
2: }
```

Das Modul erhält die Nummer der aktuellen Phase und die Gesamtanzahl an Phasen. Dadurch können sich Module je nach Phase unterschiedlich verhalten. Die `user_newPhase` Funktion wird unabhängig davon, ob das Modul während der Phase überhaupt aufgerufen werden wird, immer für alle Module aufgerufen.

Die generierte Funktion ist leer und bleibt unverändert, da der Gausskanal an dieser Stelle nichts erledigen muss.

4.4.2.9 user_serialize

Die `user_serialize` Funktion wird immer dann gerufen, wenn die Simulation ihren Zustand persistieren soll.

```
  1: Lethe::State user_serialize() {

+ 2:     unsigned char* rngState = (unsigned char*) gsl_rng_state(_rng);
+ 3:     const size_t size = gsl_rng_size(_rng);
+ 4:     _state.rngState.clear();
+ 5:     for (unsigned int i = 0; i < size; ++i) {
+ 6:         _state.rngState.push_back(rngState[i]);
+ 7:     }

  8:     return _state;
  9: }
```

Alle Zustände eines Moduls, die in keiner Bibliothek enthalten sind, sollten ständig von der `_state`-Variablen gespeichert werden. Dann kann der Entwickler einfach die generierte Implementierung dieser Funktion verwenden, die diese Variable zurück gibt (Zeile 8).

Im Fall des Gausskanals ist der Zustand allerdings im Zufallsgenerator-Objekt enthalten. Die vom Entwickler hinzugefügten Zeilen lesen diesen Zustand aus (Zeilen 2+3) und speichern ihn in der Zustandsvariablen des Moduls (Zeilen 4-7)[4], welche anschließend zurückgegeben wird (Zeile 8).

Die Variable `state` des Modulzustandes wurde vom Entwickler als Byte-Array spezifiziert. Der dazugehörige C++-Typ heißt `std::vector<unsigned char>`. Dieser Typ

[4] Eine typische C++ Implementierung würde die Standardbibliothek (vgl. [35] Teil III) zum Kopieren der Container verwenden. Damit zum Verständnis des Beispiels wenig C++ Kenntnisse nötig sind, werden die Daten auf klassische Weise kopiert.

definiert die Funktionen `clear` und `push_back`, die in den Zeilen 4 und 6 verwendet werden. Die erste Funktion entleert den Vektor und die zweite Funktion hängt ein Element hinten an.

Da die von der GSL angebotenen Möglichkeiten, den Zustand des Zufallsgenerators zu lesen und zu schreiben, leider nicht plattformunabhängig sind (vgl. [23] Kapitel 17.5 und 17.8), sollte diese Implementierung nicht in heterogenen Netzwerken eingesetzt werden. Das ist ein Beispiel für die in 3.1.9 erwähnten Einschränkungen, die die Verwendung einer numerischen Bibltiothek mit sich bringen kann.

4.4.2.10 user_deserialize

Die `user_deserialize` Funktion wird immer dann gerufen, wenn ein vorher persistierter Zustand wieder geladen werden soll

```
  1: void user_deserialize(const Lethe::State& state) {

+ 2:     unsigned char* rngState = (unsigned char*) gsl_rng_state(_rng);
+ 3:     assert(state.rngState.size() == gsl_rng_size(_rng));
+ 4:     for (unsigned int i = 0; i < state.rngState.size(); ++i) {
+ 5:         rngState[i] = state.rngState[i];
+ 6:     }

  7:     _state = state;
  8: }
```

Das Modul bekommt beim Aufruf das Zustandsobjekt, das es zuvor beim Aufruf von `user_serialize` zurückgegeben hat. Die generierte Implementierung speichert es (Zeile 7), so dass anschließend der Zustand vom Zeitpunkt des entsprechenden Aufrufes von `user_serialize` wiederhergestellt ist.

Die Implementierung des Gausskanals muss allerdings noch den Zustand des Zufallsgenerators wiederherstellen, was die hinzugefügten Zeilen 2 bis 6 erledigen.

Wie bereits erwähnt, ist das Lesen und Schreiben des Zustanden von Zufallsgeneratoren der GSL nicht portabel. Es ist unklar, wie garantiert sichergestellt werden kann, dass die Übertragung des Zustandes korrekt ist, falls die Fortsetzung auf einer anderen Plattform stattfindet. In der Zeile 3 wird überprüft, ob zumindest die Größe des Zustandes stimmt.

4.4.2.11 user_getResult

Am Ende einer Runde wird jedes Modul aufgefordert, die Ergebnisse der Simulation zu liefern.

```
  1: Lethe::Result user_getResult() {
  2:     return _result;
  3: }
```

Die generierte Implementierung liefert einfach das Ergebnisobjekt des Moduls zurück (Zeile 2). Falls die `user_trigger`-Aufrufe immer auf diesem Objekt arbeiten, kann der Entwickler die Funktion so verwenden. Ansonsten muss er in dieser Funktion die Ergebnisse geeignet ermitteln und in das `_result`-Objekt schreiben.

4.4.2.12 user_getProgress

Die `user_getProgress` Funktion wird immer gerufen, wenn der Benutzer über den Verlauf der Simulation informiert werden möchte.

```
1: float user_getProgress() {
2:     return 1;
3: }
```

Die Funktion gibt durch einen Rückgabewert zwischen 0 (0%) und 1 (100%) an, wie weit die aktuellen Phase vorangeschritten ist. Da Lethe von den Werten aller Module das Minimum nimmt, kann der Entwickler diese Funktion unverändert verwenden, wenn sein Modul keine Aussage über den Verlauf machen kann oder soll.

4.4.2.13 Hilfsfunktionen

Die Funktionen `user_serialize`, `user_deserialize`, `user_reset` und `user_getResult` verwenden modulabhängige Typen. Die Laufzeitumgebung ist allerdings generisch und kann mit diesen Typen nicht umgehen. Deshalb gibt es für jede dieser Funktionen eine Hilfsfunktion mit dem Präfix `lethe_`. Diese Hilfsfunktionen vermitteln zwischen generischen Byte-Strömen aus der Laufzeitumgebung und den typisierten Objekten des Moduls. Sie dienen damit dem Komfort des Entwicklers.

Darüberhinaus gibt es noch eine Hilfsfunktion für die `user_trigger` Funktion. Sie setzt den `caller` Parameter an Hand einer internen Kennung, die von der Laufzeitumgebung geliefert wird.

4.5 Das Scheduling

Wie die Abschnitte 4.2 und 4.4 erklären, ist die Laufzeitumgebung in zwei Schichten aufgeteilt. Die obere Schicht ist dabei verantworlich für das Scheduling. Dazu muss der Simulationsgraph aufgebaut werden, damit sich die Streamobjekte und die Modulobjekte gegenseitig kennen. Abschnitt 4.5.1 erklärt, welche Probleme sich dabei ergeben und wie diese gelöst werden. In der Diskussion der unteren Schicht blieb in Abschnitt 4.3.2 die Frage offen, woher die Ports die Streamobjekte kennen. Die Antwort auf diese Frage ergibt sich aus den Erläuterungen von Abschnitt 4.5.1.

Es existieren zwei Typen von Interaktionen zwischen den Schichten. Der erste ist die Delegation von Schedulingentscheidungen von der unteren Schicht an die obere Schicht. Dort werden ggf. implizite Aufrufe vorgenommen und überprüft, ob eine der Bedingungen für die Regeln des impliziten Beendens (vgl. Abschnitt 3.1.4) erfüllt ist. Je nachdem, wird die untere Schicht angewiesen, mit einer entsprechenden Ausnahme zu reagieren

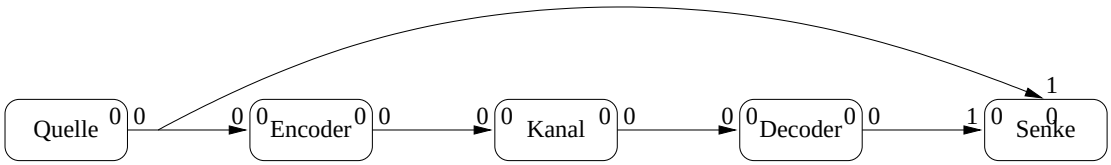


Abbildung 4.1: Der Hypergraph des Fallbeispieles mit Kennungen

(vgl. Abschnitt 4.3.2) oder den Datentransport vorzunehmen. Der zweite Typ von Interaktionen sind Auskünfte, die die obere Schicht von der unteren einfordert, um die Schedulingentscheidungen treffen zu können. Das betrifft vor allem Auskünfte über den Füllstand des Puffers.

4.5.1 Vernetzung

Ein Modul kann mit beliebig vielen Streams und ein Stream mit beliebig vielen Modulen verbunden sein (3.1.2). Wenn Streams und Module miteinander kommunizieren, brauchen sie folglich ein Adressierungsschema. Einfach ausgedrückt bedeutet das, dass die Ereignisbehandlungsfunktionen der Streams bzw. der Module einen Parameter haben, der angibt, welches Modul bzw. Stream das Ereignis ausgelöst hat.

Eine Möglichkeit ist, alle Module und alle Streams jeweils zu nummerieren. Das würde zweifelsfrei funktionieren, ist allerdings in diesem Fall ungeeignet. Der Grund dafür ist, dass die Module und die Streams verschiedene Felder besitzen, in denen sie für jeden Kommunikationspartner Informationen zum Scheduling speichern. Ein Beispiel hierfür ist die Menge der aktiven Module eines Streams. Beim obigen Adressierungsschema muss die Kennung des Partners irgendwie auf einen Index in diese Felder abgebildet werden, was sehr umständlich ist.

Deshalb verwendet Lethe ein relatives Adressierungsschema. Dabei werden pro Stream die angeschlossenen schreibenden Module durchnummeriert. Die Reihenfolge der Module entspricht dabei der Reihenfolge der dazugehörigen Ports, die sie in der Simulationsbeschreibung bei den Eigenschaften des Streams haben. Dasselbe wird für die lesenden Module gemacht. Umgekehrt gilt das auch für die eingehenden und die ausgehenden Streams.

Anschaulich heißt dies, dass die Module Kennungen wie z.B. “erster Schreiber” oder “dritter Leser”, und die Streams Kennnungen wie z.B. “zweiter ausgehender Stream” oder “erster eingehender Stream” erhalten. Die Kennung ist im Kontext eines Streams bzw. eines Moduls jeweils eindeutig. Die Eigenschaften “Schreiber” und “Leser” bzw. “eingehend” und “ausgehend” ergeben sich dabei implizit durch die Wahl der Ereignisbehandlungsfunktion. So würde z.B. niemals ein Schreiber das Ereignis auslösen, dass ein Leser beendet hat.

Abbildung 4.1 zeigt den Hypergraph aus dem Fallbeispiel mit den Kennungen der Module und Streams. So ist die Senke für den Stream zwischen Quelle und Decoder der zweite Leser. Für die Senke ist dieser Stream der erste und der Stream zwischen Decoder und Senke der zweite eingehende Stream. Diese Festlegung richten sich, wie

bereits erwähnt, nach der Simulationsbeschreibung. Die dort festgelegte Reihenfolge ist zwar willkürlich, bleibt aber bei eventuellen Migrationen oder Fortsetzungen der Simulation konstant.

Die Vernetzung der Module und der Streams findet ausschließlich im Code statt, den der Generator für jede Simulation erzeugt (vgl. Abschnitt 3.2.3). Nachdem die Modul- und die Streamobjekte erzeugt wurden, werden für jede Verbindung einmal eine Funktion des Modulobjektes und einmal eine des Streamobjektes gerufen. Die Parameter dieser Funktionen sind ein Zeiger auf den Kommunikationspartner und die beiden Kennungen, die die beiden Objekte untereinander verwenden.

Im Falle der Streams ist die gerufene Funktion zur Vernetzung in der oberen Schicht definiert, da die untere Schicht keine Referenzen auf seine Leser und Schreiber braucht. Im Falle der Module allerdings ist die gerufene Funktion in der unteren Schicht definiert. Diese Funktion gibt zum einen dem Adapter (vgl. 4.3.2) und zum anderen der oberen Schicht den Zeiger und die Kennungen.

4.5.2 Ereignisse

Neben den impliziten Aufrufen existieren noch weitere Ereignisse in der oberen Schicht. Diese werden alle durch das explizite oder implizite Beenden eines Moduls ausgelöst.

So teilt das Modul allen seinen eingehenden Streams unter Nennung der Kennung aus Abschnitt 4.5.1 mit, dass ein Leser beendet hat. Den ausgehenden Streams wird entsprechend mitgeteilt, dass ein Schreiber beendet hat. Erkennt ein Stream, dass der letzte Leser beendet hat, so teilt er allen seinen Schreibern wieder unter Nennung seiner Kennung mit, dass ein ausgehender Stream geschlossen wurde. Analoges geschieht, falls der letzte Schreiber beendet. So führen alle Module und Streams ständig Buch über den Zustand des Graphen. Sie brauchen diese Informationen, um Entscheidungen für das Scheduling treffen zu können.

Neben den Streams benachrichtigt ein Modul beim Beenden auch den Scheduler. Dieser kann daraufhin das Modul ggf. aus der Menge der Beobachter oder Protokollierer entfernen (vgl. Abschnitt 3.1.4).

4.5.3 Die call-Funktion

Die `call` Funktion ist in der abstrakten `::Core::Module`-Klasse definiert. Sie wird von den Streams und dem Scheduler gerufen, und ruft ihrerseits die `user_trigger` Funktion.

Der Aufruf der `user_trigger` Funktion ist mit einer Reihe von Ausnahmebehandlungen versehen. Egal auf Grund welcher Ausnahme die `user_trigger` Funktion verlassen wurde, löst die Funktion die in Abschnitt 4.5.2 beschriebenen Ereignisse aus. Ist die Ausahme keine, die von den Streams geworfen wird (vgl. Abschnitt 4.3.2) und damit zum normalen Ablauf der Simulation gehört, wird eine entsprechene Lognachricht an den Breeze Server (vgl. Abschnitt 3.2.6) geschickt.

Bevor die `call` Funktion die `user_trigger` Funktion ruft, legt sie ein `LoopPrevention` Objekt an. Die `LoopPrevention`-Klasse setzt in ihrem Konstruktor eine boolsche Variable, die der Destruktor wieder zurück setzt. Diese Variable ist dabei eine Membervariable der `::Core::Module`-Klasse, die angibt, ob die `call`

Funktion des Moduls bereits im Wartestack des Schedulings vorhanden ist. Falls die Variable beim Aufruf des Konstruktors bereits gesetzt ist, wirft die Klasse eine `LoopPrevention::LoopDetected` Ausnahme. Auf diese Weise wird das Verbot von zirkulären Abhängigkeiten durchgesetzt (vgl. Abschnitt 3.1.3). Durch die Verwendung des Destruktors ist garantiert, dass die boolsche Variable zurückgesetzt wird, egal aus welchem Grund die `call` Methode terminiert.

Falls eine `LoopPrevention::LoopDetected` Ausnahme geworfen wird, fängt die `call` Funktion diese selbst auf und wirft statt dessen eine `::Core::Exceptions::LoopDetected` Ausnahme, der sie den Namen des Moduls mitgibt. Diese Ausnahme folgt nun dem Wartestack des Schedulings und wird von der `call` Funktion des nächsten wartenden Moduls gefangen. Diese fügt wiederum den Namen des Moduls an die Ausnahme hinzu und wirft sie weiter. Auf diese Weise erhält der Scheduler am Ende die Ausnahme mitsamt der Folge von impliziten Aufrufen, die zu der Verklemmung führten. Der Scheduler versendet daraufhin eine entsprechende Lognachricht an den Breeze Server und beendet die Simulation.

5 Messungen

Dieses Kapitel klärt, inwiefern das Framework der Anforderung bezüglich einer hohen Ausführungsgeschwindigkeit gerecht wird. Dazu wurden im Rahmen dieser Arbeit mehrere Messungen durchgeführt, die im Folgenden dokumentiert sind.

Der den Messungen zugrundeliegende Hypergraph der Simulationen, ist in Abbildung 5.1 zu sehen. Er entspricht dem Hypergraphen des Fallbeispieles aus Abschnitt 3.1.2. Die Kanten sind durchnummeriert, damit in den folgenden Abschnitten darauf referenziert werden kann.

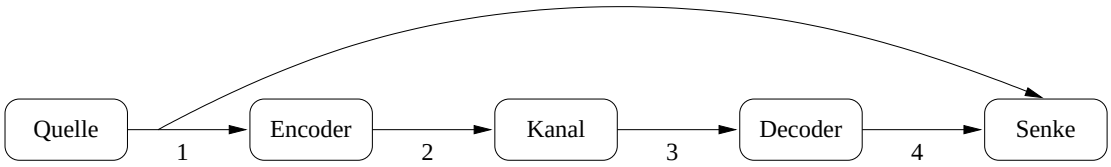


Abbildung 5.1: Die Simulation für die Messungen

Das Laufzeitverhalten der Algorithmen in den Knoten ist unabhängig von der Implementierung des Frameworks. Deshalb wird es für die folgenden Messungen nicht berücksichtigt. Stattdessen sind die Knoten so implementiert, dass sie die Daten unverändert durchreichen. Die Quelle erzeugt fortlaufend aufsteigende Zahlen und die Senke überprüft, ob die Zahlen von der Quelle und vom Decoder jeweils identisch sind. Damit ist diese Messung gleichzeitig ein Funktionstest der Laufzeitumgebung.

Der folgende Abschnitt listet die Testkriterien der durchgeführten Messungen auf. Anschließend erklärt Abschnitt 5.2, wie sie wiederholt werden können. Die restlichen Abschnitte dokumentieren die eigentlichen Messungen.

5.1 Testkriterien

Die Testumgebung hatte für alle Messungen die folgenden Eigenschaften:

Prozessor Intel Pentium M Prozessor mit 1.73GHz

Speicher 2GB

System Debian GNU/Linux [8]

Compiler GNU C++ Compiler 4.0.3 [14]

Compileroptionen -DNDEBUG -O1 (Assertions deaktivieren, einfache Optimierung)

Die Simulation endet, sobald die Senke 10 Millionen Zahlen gelesen und verglichen hat. Dies ist eine realistische Anzahl für typische Simulationen. Gemessen wird im folgenden jeweils die Zeit, die zwischen dem Start und dem Ende der ersten und einzigen Phase der Simulation vergeht.

5.2 Wiederholung der Messungen

Bei den Messungen wurden jeweils andere Implementierungen der Module verwendet. Ihre Eigenschaften und die der Kanten haben außerdem einen Einfluß auf den Code, den der Generator erzeugt hat. Damit die Messungen wiederholbar sind, existiert für jede von ihnen eine eigene Version von Lethe, die auf der beigefügten CD enthalten ist. Tabelle 5.1 listet diese auf:

Abschnitt	Version
5.3	342
5.4	345
5.4.1	353
5.4.2	354
5.5	355
5.5.1	363-branch
5.6	355 und 363-branch

Tabelle 5.1: Die verwendeten Softwareversionen

5.3 Verwaltungsaufwand

Idealerweise sollte nur ein Bruchteil der Laufzeit einer Simulation für den Verwaltungsaufwand verbraucht werden. Die Messung überprüft, ob das für Lethe zutrifft, indem sie die Laufzeit der Simulation bestimmt, bei der, wie bereits erwähnt, in den Knoten minimale Laufzeit verbraucht wird.

Die Eigenschaften der Knoten sind so gewählt, dass an keiner Stelle benutzerdefinierte Konvertierungen auftreten (vgl. Abschnitt 4.2.4). Da ein `Complex` aus zwei Fließkommazahlen und ein `Integer` aus einer Ganzzahl besteht, kommt es im Encoder und Decoder unvermeidlich zu Konvertierungen zwischen den Typen `int` und `double`. Der Aufwand dafür ist aber vernachlässigbar.

Die für die Messungen relevanten Eigenschaften der Ports sind in Tabelle 5.2 aufgelistet. Tabelle 5.3 zeigt die Eigenschaften der Kanten.

Die Messung ergab eine Laufzeit von 57 Sekunden. Erfahrungsgemäß hängt die Laufzeit einer Simulation sehr stark von der Komplexität des simulierten Codes ab. So ergeben sich für Simulationen diesen Umfangs Zeiten zwischen einer Stunde und einem Tag. Der Anteil der Verwaltung liegt also zwischen 1,58% und 0,07% und ist damit für komplexe Codes vernachlässigbar klein.

Knoten	Kante	Basistyp	Arbeitstyp	Blockgröße
Quelle	1	`Integer`	`Integer`	1
Encoder	1	`Integer`	`Integer`	2
	2	`Complex`	`Complex`	1
Kanal	2	`Complex`	`Complex`	1
	3	`Complex`	`Complex`	1
Decoder	3	`Complex`	`Complex`	1
	1	`Integer`	`Integer`	2
Senke	1	`Integer`	`Integer`	1
	3	`Integer`	`Integer`	1

Tabelle 5.2: Die Eigenschaften der Ports aus der Messung 5.3

Kante	1	2	3	4
Puffertyp	`Integer`	`Complex`	`Complex`	`Integer`
Basistyp	`Integer`	`Complex`	`Complex`	`Integer`

Tabelle 5.3: Die Eigenschaften der Kanten aus der Messung 5.3

5.4 Konvertierungen

Diese Messung zeigt, wieviel Laufzeit durch Konvertierungen verbraucht wird. Dazu verwendet der eingehende Port des Kanals den Arbeitstyp `gsl_complex`. Damit kommt es bei Lese- und Schreibzugriffen des Kanals zu benutzerdefinierten Konvertierungen.

In der Tabelle 5.4 sind die Eigenschaften der Ports aufgelistet, wobei die Unterschiede zur letzten Messung (vgl. Tabelle 5.2) hervorgehoben sind. Die Eigenschaften der Kanten sind bezüglich der letzten Messung unverändert (vgl. Tabelle 5.3).

Knoten	Kante	Basistyp	Arbeitstyp	Blockgröße
Quelle	1	`Integer`	`Integer`	1
Encoder	1	`Integer`	`Integer`	2
	2	`Complex`	`Complex`	1
Kanal	2	`Complex`	**`gsl_complex`**	1
	3	`Complex`	`Complex`	1
Decoder	3	`Complex`	`Complex`	1
	1	`Integer`	`Integer`	2
Senke	1	`Integer`	`Integer`	1
	3	`Integer`	`Integer`	1

Tabelle 5.4: Die Eigenschaften der Ports aus der Messung 5.4

Erstaunlicherweise verursachten die Konvertierungen keinen messbaren Mehraufwand. Dies deutet darauf hin, dass der Aufwand für die restliche Verwaltung deutlich überwiegt. Die folgende Messung untersucht dies näher.

5.4.1 Erhöhung der Konvertierungshäufigkeit

Ausgehend von der vorherigen Messung wird in einem zweiten Schritt die Anzahl der Konvertierungen maximiert, indem jeder Port einen vom Basistyp verschiedenen Arbeitstyp erhält und die Optimierung aus Abschnitt 4.2.5 ausgeschaltet ist. Die Tabelle 5.5 veranschaulicht dies, wobei sich die hervorgehobenen Unterschiede auf die vorhergehende Messung beziehen (vgl. Tabelle 5.4). Die Eigenschaften der Kanten sind weiterhin unverändert (vgl. Tabelle 5.3).

Knoten	Kante	Basistyp	Arbeitstyp	Blockgröße
Quelle	1	`Integer`	**`int`**	1
Encoder	1	`Integer`	**`int`**	2
	2	`Complex`	**`gsl_complex`**	1
Kanal	2	`Complex`	`gsl_complex`	1
	3	`Complex`	**`gsl_complex`**	1
Decoder	3	`Complex`	**`gsl_complex`**	1
	1	`Integer`	**`int`**	2
Senke	1	`Integer`	**`int`**	1
	3	`Integer`	**`int`**	1

Tabelle 5.5: Die Eigenschaften der Ports aus der Messung 5.4.1

Diesesmal konnte eine Erhöhung der Laufzeit um 4 Sekunden auf 61 Sekunden gemessen werden. Für eine Berechnung der zusätzlichen Laufzeitkosten, die durch Konvertierungen in einem Port entstehen, ist die Messgenauigkeit zu niedrig. Es lässt sich nur abschätzen, dass für die 10 Millionen Konvertierungen pro Port weniger als eine halbe Sekunde benötigt wurde.

5.4.2 Verwendung der Optimierung

Die gleiche Messung (vgl. Tabelle 5.5) mit eingeschalteter Optimierung ergibt die Eigenschaften für die Kanten aus Tabelle 5.6. In dieser Messung besitzen sie zum ersten

Kante	1	2	3	4
Puffertyp	**`int`**	**`gsl_complex`**	**`gsl_complex`**	**`int`**
Basistyp	`Integer`	`Complex`	`Complex`	`Integer`

Tabelle 5.6: Die Eigenschaften der Kanten aus der Messung 5.4.2

Mal einen vom Basistyp abweichenden Arbeitstyp, was in diesem Fall dazu führt, dass keine benutzerdefinierten Konvertierungen nötig sind. Es ist also zu erwarten, dass sich dieselbe Laufzeit wie in der Messung aus Abschnitt 5.3 ergibt, was durch die Messungen bestätigt wurde.

5.5 Erhöhung der Blockgröße

Verwenden die Knoten größere Blockgrößen, dann kommt es bei der gleichen Datenmenge insgesamt zu weniger Schreib- und Leseaufrufen, wodurch der Verwaltungsaufwand sinkt. Die folgende Messreihe prüft, in welchem Rahmen sich die Änderung des Verwaltungsaufwandes dabei bewegen kann.

Dazu wird die Implementierung der Knoten dahingehend verändert, dass sie über die Vektor-Objekte, die nun große Blöcke von Daten enthalten, iterieren und die Daten einzeln wie bisher durchreichen. Zu jeder Blockgröße wird anschließend die Laufzeit der Simulation gemessen.

Die Eigenschaften der Ports sind in in Tabelle 5.7 abgebildet. Der Paramter b ist die Variable für die wachsende Blockgröße. Die Optimierung ist eingeschaltet, so dass sich für die Eigenschaften der Kanten die Tabelle 5.8 ergibt.

Knoten	Kante	Basistyp	Arbeitstyp	Blockgröße
Quelle	1	`Integer`	`int`	$1 \cdot b$
Encoder	1	`Integer`	`int`	$2 \cdot b$
	2	`Complex`	`gsl_complex`	$1 \cdot b$
Kanal	2	`Complex`	`gsl_complex`	$1 \cdot b$
	3	`Complex`	`gsl_complex`	$1 \cdot b$
Decoder	3	`Complex`	`gsl_complex`	$1 \cdot b$
	1	`Integer`	`int`	$2 \cdot b$
Senke	1	`Integer`	`int`	$1 \cdot b$
	3	`Integer`	`int`	$1 \cdot b$

Tabelle 5.7: Die Eigenschaften der Ports aus der Messung 5.5

Kante	1	2	3	4
Puffertyp	`int`	`gsl_complex`	`gsl_complex`	`int`
Basistyp	`Integer`	`Complex`	`Complex`	`Integer`

Tabelle 5.8: Die Eigenschaften der Kanten aus der Messung 5.5

Die Abbildungen 5.2 und 5.3 zeigen die Ergebnisse dieser Messreihe. Wie erwartet sinkt der Verwaltungsaufwand mit steigender Blockgröße.

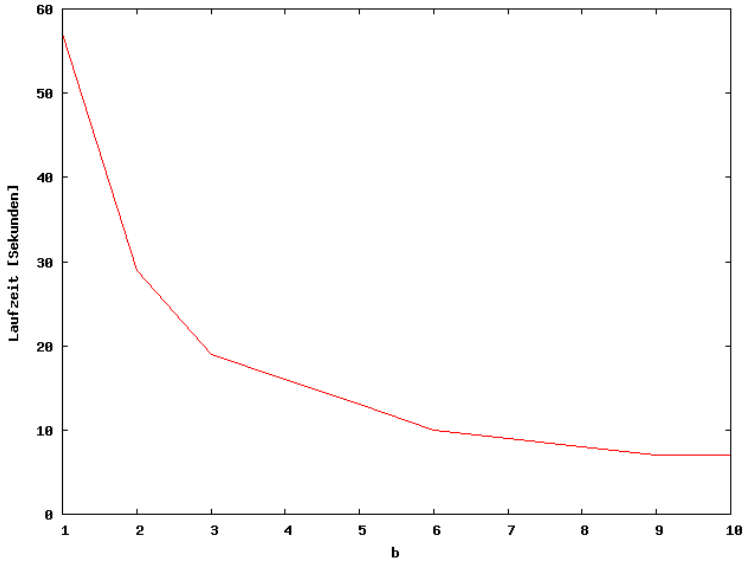


Abbildung 5.2: Der Verwaltungsaufwand in Abhängigkeit zur Blockgröße

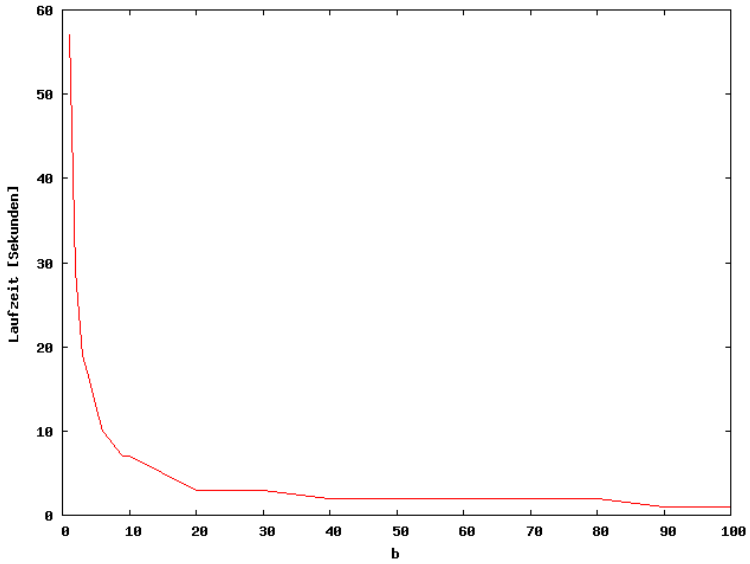


Abbildung 5.3: Der Verwaltungsaufwand in Abhängigkeit zur Blockgröße - größere Skala

Die Itereration über die Vektor-Objekte erfordert eine einmalige Erhöhung der Codekomplexität in den Knoten. Dafür kann die Laufzeit im Austausch gegen Speicher gesenkt werden. Der Anteil der Verwaltung an der Laufzeit geht asymptotisch gegen null. Ab einer hunderfachen Blockgröße liegt die Laufzeit im Bereich von einer Sekunde und damit unterhalb der Messgenauigkeit.

5.5.1 Deaktivierung der Optimierung

Wenn der Verwaltungsaufwand, wie im letzten Abschnitt gezeigt, gegen null gehen kann, dann lässt sich durch eine weitere Messreihe vielleicht genauer feststellen, wie groß der Anteil der Konvertierungen an der Laufzeit ist. Denn die Anzahl der Konvertierungen ist unabhängig von der Blockgröße.

In dieser Messungreihe wird deshalb die Optimierung ausgeschaltet. Die Eigenschaften der Ports entspricht denen aus Tabelle 5.7. Durch die ausgeschaltete Optimierung ergeben sich für die Kanten wieder die Eigenschaften der ersten Messung aus diesem Kapitel (vgl. Tabelle 5.3).

Die Ergebnisse dieser Messreihe sind in den Abbildungen 5.4 und 5.5 zu sehen. Zum Vergleich sind die Ergebnisse der vorherigen Messreihe mit eigenschalteter Optimierung mit eingezeichnet.

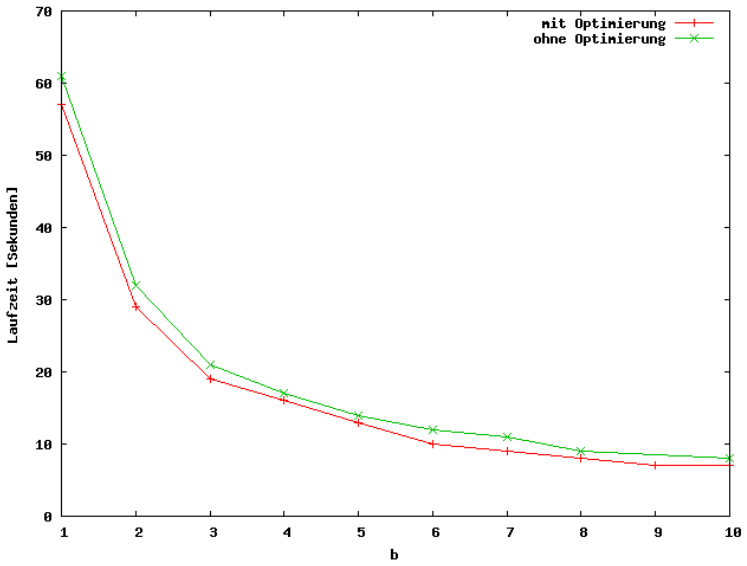


Abbildung 5.4: Vergleich mit und ohne Optimierung

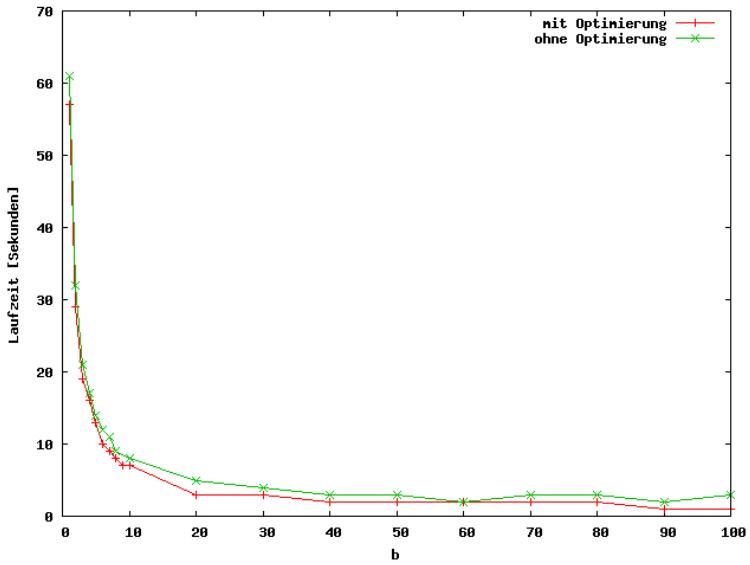


Abbildung 5.5: Vergleich mit und ohne Optimierung - größere Skala

Die Laufzeiten, die sich ohne Optimierung ergeben, bleiben wie erwartet über denen mit der Optimierung. Allerdings fällt die Differenz zwischen ihnen mit der steigenden Blockgröße, was nicht erwartet wurde. Eine mögliche Begründung für dieses Verhalten ist, dass dieses Programm auf Grund der Iterationen eine höhere Lokalität des Codes und der Daten besitzt, so dass die CPU Caches besser genutzt werden können.

5.6 Erhöhung der Datenmenge

Bei den bisherigen Messungen blieb der Laufzeitgewinn durch die Optimierung sehr gering. Da aufgrund dieser Optimierung die Modul-Klassen Templates sein müssen (vgl. Abschnitt 4.4) und sie somit negative Auswirkungen auf die Entwickler API hatte, wäre es in Anbetracht dieser Messergebnisse vielleicht besser gewesen, sie nicht einzuführen.

Es ist allerdings möglich, dass die bisher verwendete Datenmenge von 10 Millionen übertragenen Zahlen zu gering ist, als dass sich der Effekt der Optimierung bei den Messungen herauskristallisieren kann. Deshalb wird in einer abschließenden Messreihe die Datenmenge erhöht und die Laufzeit jeweils mit und ohne Optimierung gemessen. Es ist in beiden Fällen ein lineares Wachstum zu erwarten. Von Interesse wird sein, wie groß die jeweiligen Steigungen sind.

Die Eigenschaften der Ports sind in beiden Fällen die aus Tabelle 5.9. Für die Messung wurde die hunderfache Blockgröße gewählt, da sich dadurch, wie in Abschnitt 5.5 erläutert, der Aufwand für die Konvertierungen deutlicher zeigt. Die Kanten besitzten für den Fall mit Optimierung die Eigenschaften aus Tabelle 5.6 und für den Fall ohne Optimierung die Eigenschaften aus Tabelle 5.3.

Abbildung 5.6 zeigt die Ergebnisse dieser Messreihe. In beiden Fällen wächst die Laufzeit ungefähr linear mit der Anzahl der übertragenen Zahlen. Die Steigungen sind

Knoten	Kante	Basistyp	Arbeitstyp	Blockgröße
Quelle	1	`Integer`	`int`	**100**
Encoder	1	`Integer`	`int`	**200**
	2	`Complex`	`gsl_complex`	**100**
Kanal	2	`Complex`	`gsl_complex`	**100**
	3	`Complex`	`gsl_complex`	**100**
Decoder	3	`Complex`	`gsl_complex`	**100**
	1	`Integer`	`int`	**200**
Senke	1	`Integer`	`int`	**100**
	3	`Integer`	`int`	**100**

Tabelle 5.9: Die Eigenschaften der Ports aus der Messung 5.5

allerdings verschieden: mit der Optimierung beträgt sie 0.14 und ohne die Optimierung 0.23 Sekunden pro 10 Millionen Zahlen. Die Optimierung bewirkt damit nahezu eine Halbierung der Laufzeit.

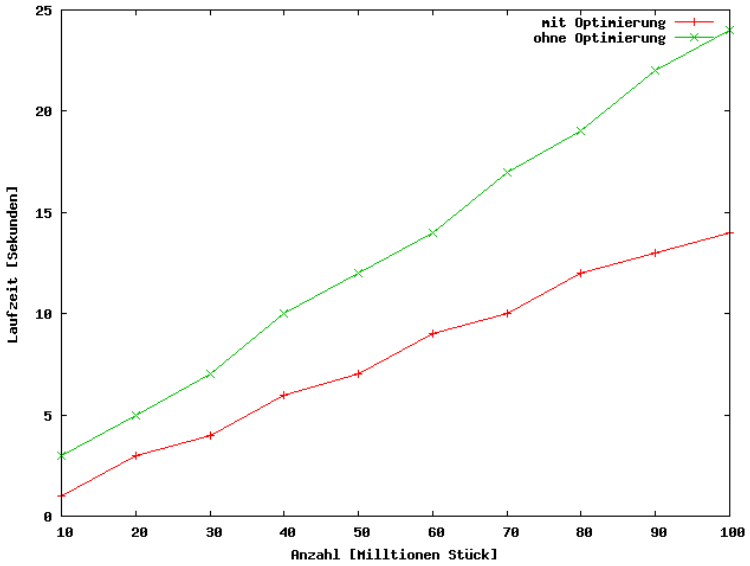


Abbildung 5.6: Erhöhung der Datenmenge - Vergleich mit und ohne Optimierung

Ob die Optimierung letztendlich sinnvoll ist oder nicht, kann nicht einfach entschieden werden. Relativ gesehen bewirkt sie signifikante Einsparungen für den Verwaltungsaufwand, jedoch absolut gesehen sind das nur ein paar Sekunden, die verglichen mit den langen Simulationszeiten keine Rolle spielen.

6 Schlußfolgerungen und Ausblicke

Dieses Kapitel liefert eine abschließende Diskussion über Lethe. Zuerst erörtert Abschnitt 6.1, ob das Framework die in Kapitel 1 gestellten Anforderungen erfüllt. Da Lethe speziell für den Einsatz in der Kanalcodierung entwickelt wurde, stellt sich die Frage, welche Anwendungsgebiete es letztendlich abdeckt. Diese Frage klärt Abschnitt 6.2. Abschnitt 6.3 erläutert anschließend, welche lizenzrechtlichen Konsequenzen die GPL für Lethe und dessen Anwender mit sich bringt. Zuletzt nennt Abschnitt 6.4 Ideen, die im Rahmen dieser Arbeit aufgekommen sind und aus Zeitgründen nicht umgesetzt wurden. Sie sind mögliche Punkte für die Weiterentwicklung von Lethe.

6.1 Erfüllung der Anforderungen

In den Entwurf und die Implementierung von Lethe sind Erkenntnisse aus Gesprächen mit potentiellen Nutzern eingeflossen, um eine möglichst praxistaugliche Schnittstelle zu erhalten. Zu den Konsequenzen dieser Gespräche gehören der `caller`-Parameter der Behandlungsfunktionen (vgl. Abschnitt 4.4.2.4) und die Möglichkeit einer Vorausschau auf eingehende Daten (vgl. Abschnitt 4.3.2).

Trotz sorgfältiger Planung und Entwicklung wird erst der praktische Einsatz von Lethe zeigen, ob die Entwickler API ausgereift genug ist, um alle Anwendungsfälle abzudecken und ob sie intuitiv bedienbar ist. Die bisherigen Gespräche und erste Versuche von Nutzern deuten allerdings daraufhin, dass diese Anforderungen erfüllt sind. Die anderen Anforderungen an die Entwickler API sind// sicher erfüllt. Erstens zeigen die Messungen in Kapitel 5, dass der Anteil der Verwaltung an der Laufzeit einer Simulation vernachlässigbar ist. Zweitens führen die vordefinierten Funktionen (vgl. Abschnitt 4.4.2) und das implizite Beenden (vgl. Abschnitt 3.1.4) dazu, dass das Sinnvollste geschieht, wenn der Entwickler nicht eingreift und drittens verhindern die Hilfsfunktionen (vgl. Abschnitt 4.4.2.13), dass der Entwickler Code schreiben muss, dessen Zweck und Hintergründe nicht offensichtlich sind.

6.2 Anwendungsgebiete

Lethe wurde speziell auf den Einsatz in der Kanalcodierung hin entwickelt. Das bedeutet aber nicht, dass es nicht auch in anderen Anwendungsgebieten eingesetzt werden kann.

Jedes Problem, dass sich als Hypergraph modellieren lässt, bei dem in den Knoten stationäre Algorithmen Daten von eingehenden auf ausgehende Kanten abbilden, lässt sich mit Lethe simulieren. Das bedeutet insbesondere, dass Lethe nicht nur in der Kanalcodierung, sondern in vielen Gebieten der Informationstheorie eingesetzt werden kann.

Durch das speziallisierte Scheduling (vgl. Abschnitt 3.1.3) eignet sich Lethe nur bedingt für die Simulation von verteilten Algorithmen, wie z.B. Netzwerkprotokollen. Insbesondere das Fehlen einer Simulationszeit ist dabei das Problem. So lässt sich z.B. die Verzögerung von Nachrichten nur umständlich realisieren. Auch für Simulationen von Echtzeitsystemen ist Lethe deshalb keine gute Wahl. Dazu kommt in diesem Fall, dass Lethe keine Abbdildung von Interrupts ermöglicht.

6.3 Lizenzrecht

Lethe verwendet die "Internet Communication Engine" (ICE [21]). Diese wird von der Firma ZeroC unter der "GNU General Public Licence" (GPL [16]) vertrieben. Unter anderem schreibt diese Lizenz vor, dass Programme, die GPL lizenzierten Quellcode verwenden oder gegen GPL lizenzierte Bibliotheken linken, auch unter die GPL gestellt werden müssen. Eine der Konsequenzen daraus ist, dass man den vollständigen Quellcode zur Verfügung stellen muss, wenn man sein Programm weitergibt.

Für die Weiterentwicklung von Lethe bedeutet das, dass Verbesserungen oder Erweiterungen der Allgemeinheit zur Verfügung stehen, sobald sie an einen Dritten weitergegeben werden. Dagegen sind die Möglichkeiten einer kommerziellen Nutzung von Lethe durch die GPL eingeschränkt. Da ZeroC bei Bedarf allerdings auch kommerzielle Lizenzen vergibt und das Framework außer ICE keine andere Software verwendet, ist eine kommerzielle Lizenz für diesen Teil von Lethe denkbar. Ob dies für den Rest von Lehte auch gilt, klärt [41].

Der von Lethe generierte Code steht derzeit auch unter der GPL. Da alle Module diesen Code verwenden, müssen sie auch unter der GPL stehen. Das bedeutet für einen Entwickler, dass er den Quellcode zur Verfügung stellen muss, wenn er ein Modul weiter gibt. Unterläßt er dies, spielt die GPL für ihn keine Rolle. Insofern ist es möglich, Lethe für die kommerzielle Entwicklung neuer Algorithmen zu verwenden. Verwendet das Modul eine numerische Bibliothek, deren Lizenz nicht GPL-kompatibel ist, ist die Weitergabe problematisch. Dann muss im Einzelfall geklärt werden, welche juristischen Möglichkeiten existieren. Eventuell wird in Zukunft auch eine andere Lizenz für den generierten Code festgelegt. Dann wäre es möglich, Module zu erstellen, die nicht unter der GPL stehen. Die juristischen und kommerziellen Diskussionen dazu sind allerdings komplex und werden im Rahmen dieser Arbeit deshalb nicht geführt.

6.4 Ausblicke

Lethe ist ein sehr umfangreiches Projekt. Selbst die Aufteilung auf zwei Diplomarbeiten kann nicht verhindern, dass nicht alle Probleme im Rahmen dieser Arbeiten gelöst werden können.

Insbesondere ist die Implementierung nur ein Prototyp. Für die Produktreife müssen vor allem noch vollständige Modultests gemacht werden. Auch die Robustheit der Software muss verbessert werden. An vielen Stellen bricht die Laufzeit in einem Fehlerfall einfach ab, anstatt diesen geeignet zu behandeln.

Die ersten beiden der folgenden Abschnitte benennen Punkte, die in dieser Arbeit bereits im Entwurf oder der Implementierung aufgekommen sind. Darauf folgen Abschnitte mit Ideen, die den Komfort für den Benutzer steigern.

6.4.1 IceStorm

Wie Abschnitt 3.2.6 erklärt, ist der Breeze Server aus Zeitmangel entstanden. Der Hauptnachteil gegenüber der ursprünglich gedachten Lösung, dem IceStorm Dienst zu verwenden ist, dass der Benutzer regelmäßig nach neuen Lognachrichten fragen muss.

Sollte sich herausstellen, dass ein *Pushdienst* erhebliche Vorteile mit sich bringen würde, so müsste man die ursprünglich gedachte Lösung umsetzen und den IceStorm Dienst so erweitern, dass er Nachrichten persistiert und auf Anfrage, ähnlich wie der Breeze Server, wieder zur Verfügung stellt.

6.4.2 Fehlersuche

Wie Abschnitt 3.2.3 erklärt, wird für jede Simulation ein eigenes Programm erstellt. Fehler des Benutzers oder des Entwicklers können dabei zu Compiler- oder Linkerfehlern führen. Der Benutzer muss dann die Fehlersuche anhand der Fehlermeldungen vornehmen.

Teilweise lässt sich diese Situation verhindern, wenn das Anwenderprogramm frühzeitig eine Analyse der Benutzereingaben vornimmt. Dabei besteht die Möglichkeit, dem Benutzer mitzuteilen, an welcher Stelle ihm Fehler unterlaufen sind. Wenn ein Entwickler allerdings etwas bei der Implementierung einer Konvertierungsfunktion (vgl. Abschnitt 4.2.4) oder eines Moduls (vgl. Abschnitt 4.4.2) falsch macht, kann das erst vom Compiler erkannt werden.

Eine Möglichkeit, die Situation zu verbessern ist, ein Nachschlagewerk mit den gängigsten Benutzer- und Entwicklerfehlern inklusive der dazugehörigen Compiler- oder Linkerfehlermeldung zu erstellen. Dazu muss ein geeigneter Weg der Darstellung der Fehlermeldungen gefunden werden, damit der Benutzer, ausgehend von einer konkreten Fehlermeldung, möglichst schnell Hinweise auf den Grund finden kann. Problematisch an diesem Ansatz ist, dass für jeden Compiler, vielleicht sogar für jede Compilerversion ein eigenes Nachschlagewerk gebraucht wird.

Ein komfortablere Lösung ist, wenn aus den Fehlermeldungen automatisiert eine verständliche Erklärung des Problems erzeugt wird. Programme wie z.B. STLFilt [34] zeigen, dass das möglich ist. Allerdings muss auch hierbei für jeden Compiler in jeder Version extra Arbeit gemacht werden.

Zufriedenstellend sind also beide Lösungen nicht. Vielleicht gibt es bessere Möglichkeiten, was noch untersucht werden muss.

6.4.3 Unterstützung von manueller Parallelisierung

Wie Abschnitt 3.1.5 diskutiert, parallelisiert Lethe ausschließlich an Rundengrenzen. Der Grund dafür ist, dass Lethe selbständig keine korrekte Aufspaltung einer Runde finden kann.

Falls der Benutzer eine solche Aufspaltung findet, muss er die Parallelisierung von Hand realisieren. Dazu muss er mehrere, für Lethe unabhängige Simulationen erzeugen und die Ergebnisse der einzelnen Teile mit Hilfe von externen Programmen zusammenühren.

Sollte sich herausstellen, dass die Parallelisierung von Runden einen erheblichen Zeitgewinn darstellt, ist darüber nachzudenken, ob der oben beschriebene Vorgang nicht automatisiert werden kann. Dazu muss der Benutzer angeben, ob und wie eine Runde geteilt werden kann. Zusätzlich muss die Entwickler API um eine Funktion erweitert werden, die aus einer Menge von Teilergebnissen das Gesamtergebnis eines Knotens berechnet. Lethe kann dann die Runden automatisch verteilen, am Ende der Simulationen die Teilergebnisse sammeln und sie einer der Simulationen zum Berechnen der Gesamtergebnisse geben. Neben dem Komfort für den Benutzer bringt diese Lösung den Vorteil, dass die Algorithmen zum Zusammenführen von Teilergebnissen innerhalb von Lethe und nicht in irgendwelchen, untereinander inkompatiblen Formaten für externe Programme definiert sind.

6.4.4 Unterstützung für Matlab

Wie Abschnitt 4.1 diskutiert, muss der Entwickler die Algorithmen in C++ implementieren. Die in Abschnitt 2.5 erwähnten Probleme dieser Sprache machen diese Arbeit allerdings mühsam. Wie sich aus Gesprächen ergeben hat, sind deshalb nicht alle potentiellen Entwickler bereit, Lethe zu benutzen. Dies würde sich ändern, wenn es möglich wäre, Module in Matlab zu programmieren.

Um diese Anwender zu gewinnen, muss Lethe also eine Möglichkeit bieten, Module in Matlab zu implementieren. Mit Hilfe eines Matlab-nach-C++-Compilers wäre es möglich, diese Module in die Simulation mit einzubinden.

Es wäre also zu untersuchen, ob eine Abbildung der Lethe API auf eine Matlab API möglich ist und wie man eine solche Zwischenschicht am besten realisisert. Insbesondere muss dazu geklärt werden, wie eine Abbildung der Verwaltungsklassen (vgl. Abschnitt 4.4.1) aussehen kann.

Eine weitere zentrale Frage ist, ob der gesamte Zustand eines in Matlab geschriebenen Moduls überhaupt erfassbar ist. Falls eine Matlab Bibliothek keinen Zugriff auf interne Zustände gewährt, ist das ein Problem. Denn eine vollständige Erfassung des Zustandes ist entscheidend für die Funktionalität von Lethe (vgl. Abschnitt 3.1.9).

Wichtig für eine solche Zwischenschicht ist die Frage, wieiel Laufzeit durch sie verloren geht. Immerhin wird sie während einer Simulation sehr oft durchlaufen. Sind die Kosten zu hoch, kann sich daraus je nach Anforderung ergeben, dass die Matlabunterstützung unbrauchbar ist.

Alles in allem erfordert die Unterstützung von Matlab zwar einigen Aufwand, scheint aber prinzipiell möglich zu sein.

6.4.5 Beschleunigung des Generators

Der Generator erhält den Quellcode, der zur Ausführung kommen soll, von den Source Services (vgl. Abschnitt 3.2.4). Die momentane Implementierung des Generators beginnt

für jede Simulation mit einem leeren Verzeichnis, in das die Source Services den Quellcode kopieren. Deshalb müssen die Laufzeitumgebung und alle verwendeten Module und Typen jedesmal übersetzt werden.

Die Übersetzungszeit ist verglichen mit der Laufzeit der Simulation kurz. Allerdings muss der Benutzer diese Zeit abwarten, um zu wissen, ob seine Simulation erfolgreich startet. Sollte sich herausstellen, dass diese Zeit inakzeptabel lang ist, so kann der Generator folgendermaßen beschleunigt werden:

Die Source Services unterstützen eine differentielle Übertragung von Daten. Dadurch werden im Zielverzeichnis des Generators nur die Dateien überschrieben, die der Source Service in einer anderen Version vorliegen hat. Das Buildsystem übersetzt anschließend nur die Teile neu, die sich verändert haben. Dadurch kann in den meisten Fällen die Neuübersetzung der Laufzeitumgebung und der verwendeten Module und Typen vermieden werden, da üblicherweise alle Benutzer die jeweils aktuelleste Softwareversion verwenden. Verwendet ein Benutzer eine andere Version, kann diese Lösung im schlechtesten Fall dazu führen, dass dennoch alles neu übersetzt werden muss. Eventuell wäre es denkbar, dass der Generator mehrere Arbeitsverzeichnisse verwendet, um diesen Fall zu verhindern.

7 Zusammenfassung

Das Ziel dieser Diplomarbeit war es, in Kooperation mit Stephanie Wist ein Simulationsprogramm zu schaffen, das im Gegensatz zu allen bekannten Alternativen die Möglichkeiten, die sich durch verteiltes Rechnen ergeben, vollständig ausschöpft. In einem Netzwerk von mehreren Computern kann vor allem die Wartezeit auf Ergebnisse durch paralleles Ausführen einer Simulation deutlich gesenkt werden. Zusätzlich erlaubt die Flexibilität, die sich durch die Migration von laufenden Simulationen auf andere Computer ergibt, eine optimale Nutzung der zur Verfügung stehenden Ressourcen. Zusammen mit der Erstellung von Sicherungspunkten, mittels denen eine abgebrochene Simulation fortgesetzt werden kann, bietet die Migration außerdem Sicherheit gegen Hardwareausfälle. Das Besondere an Lethe ist nicht nur, dass es alldiese Möglichkeiten bietet, sondern, dass dies alles automatisiert geschieht und der Benutzer sich dadurch auf die wesentlichen Probleme konzentrieren kann.

Das Framework, das Thema dieser Arbeit, ermöglicht diese Fähigkeiten von Lethe. Es erzielt eine hohe Ausführungsgeschwindigkeit der Simulationen und ist zugleich komfortabel zu bedienen. Das Modell des Hypergraphen verspricht außerdem mächtig genug zu sein, um alle Anwendungsfälle der Kanalcodierung abzudecken. Zuletzt ist durch die Plattformunabhängigkeit, die Unterstützung nahezu beliebiger numerischer Bibliotheken und die freie Lizenz der potentielle Anwenderkreis sehr groß. Das einzige, was diesen einschränkt, ist die Tatsache, dass der Entwickler in C++ programmieren muss.

Der Scheduler nützt die Tatsache aus, dass Lethe auf Anwendungen in der Kanalcodierung spezialisiert ist. Das ermöglicht ihm, schlicht und dennoch effizient zu sein. Allerdings gibt er keine Garantien über die Aufrufreihenfolge der Knoten, so dass sich nicht voraussagen lässt, wie eine Simulation im Detail ablaufen wird. In der Konsequenz ist die Entwicklerschnittstelle so definiert, dass diese Details im Normalfall irrelevant sind. Darüberhinaus maximiert sie die Wiederverwendbarkeit der Module, so dass sie sich in vielen verschiedenen Simulationen verwenden lassen.

Die Parallelisierung einer Simulation findet an den Grenzen der voneinander unabhängigen Runden statt und ist somit trivial. Nichtsdestotrotz wird die Wartezeit auf die Simulationsergebnisse dadurch deutlich gesenkt. Für jede Runde existiert ein Simulationsobjekt im verteilten System. Aus Robustheitsgründen sind alle Simulationsobjekte unabhängig voneinander. Die Proxies erhöhen die Robustheit zusätzlich, indem sie die Simulationen überwachen und bei Bedarf kontrolliert beenden. Außerdem kann der Benutzer bei einem Abbruch einer Simulation durch sie den Grund dafür erfahren. Zuletzt existiert noch ein zentraler Logging Server, der es dem Benutzer auf einfache Weise ermöglicht, über Ereignisse in seinen Simulationen informiert zu werden.

Der praktische Einsatz von Lethe wird zeigen müssen, inwiefern manche Vermutungen, die beim Entwurf des Frameworks gemacht wurden, zutreffen. Bisher deuten erste Gespräche mit potentiellen Nutzern darauf hin. Dennoch kann das für die Weiterent-

wicklung von Lethe bedeuten, dass nachträgliche Änderungen des Entwurfs nötig sind. Eine mit Sicherheit noch anstehende Aufgabe ist, den Prototypen, der im Rahmen dieser Arbeit implementiert wurde, zur Produktreife zu führen.

Anhang A

GNU PTH

GNU PTH steht für die "GNU Portable Threads" Bibliothek. Sie stand in der Diskussion, für das Multitasking in Lethe eingesetzt zu werden (vgl. 2.3). In Abschnitt A.1 ermittelt eine Messung die Laufzeitkosten, die durch den Kontextwechsel zwischen zwei Threads entstehen. Abschnitt A.2 nennt anschließend eine Möglichkeit, die Anzahl der Kontextwechsel gegenüber der aktuellen PTH Implementierung zu halbieren.

A.1 Zeitmessung der GNU PTH

Die folgende Zeitmessung zu den Kosten von Userland Threads wurde im Rahmen dieser Arbeit durchgeführt. Das Ergebnis war ausschlaggebend für die Entscheidung, die GNU PTH nicht zu verwenden. Die beiden folgenden Programme messen, wieviel Zeit für 1 Milliarde Kontextwechsel benötigt wird. Das erste Programm verwendet dazu die PTH Threads. Das zweite Programm verwendet statt dessen einen Verteiler und Behandlungsfunktionen.

A.1.1 Verwendung der GNU PTH

```
// swap.c
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>

ucontext_t uctx[3];

#define STACKSIZE 64 * 1024

void func1(void* arg)
{
    int i;
    int count = *((int*) arg);

    for (i=0; i<count; i++) {
        swapcontext(&uctx[0], &uctx[1]);
    }
}
```

```
void func2(void* arg)
{
    while(1) {
        swapcontext(&uctx[1], &uctx[0]);
    }
}

typedef void(*handler_t)(void*);
handler_t handler[2];

int main(int argc, char* argv[])
{
    int i;
    if (argc != 2) {
        printf("usage: %s count\n", argv[0]);
        return 1;
    }

    int count = atoi(argv[1]);

    for (i=0; i<3; i++) {
        getcontext(&uctx[i]);
    }

    handler[0] = func1;
    handler[1] = func2;

    for (i=0; i<2; i++) {
        uctx[i].uc_stack.ss_sp = malloc(STACKSIZE);
        uctx[i].uc_stack.ss_size = sizeof(STACKSIZE);
        makecontext(&uctx[i], (void(*)(void)) handler[i], 1, (void*)&count);
    }

    uctx[0].uc_link = &uctx[2];
    uctx[1].uc_link = NULL;

    swapcontext(&uctx[2], &uctx[0]);
    return 0;
}
```

A.1.2 Verwendung eines Verteilers

```
// disptacher.c
#include <stdio.h>
```

```
#include <stdlib.h>

int func1()
{
    return 0;
}

int func2()
{
    return 0;
}

int main(int argc, char* argv[])
{
    int i;

    if (argc != 2) {
        printf("usage: %s count\n", argv[0]);
        return 1;
    }

    int count = atoi(argv[1]);
    for (i=0; i<count; i++) {
        func1();
        func2();
    }
}
```

A.1.3 Umgebung

Compiler	gcc 4.0.2
Compileroptionen	-O0 (keine Optimierung)
System	Debian GNU/Linux sarge
CPU	AMD Athlon(tm) 64 Processor 3200+

A.1.4 Ergebnisse

```
$ time ./swap 1000000000
real    12m30.041s
user    7m27.952s
sys     4m32.666s

$ time ./dispatcher 1000000000
real    0m8.170s
user    0m8.167s
```

```
sys     0m0.002s
```

A.2 Halbieren der Anzahl der Kontextwechsel

Wenn bei der GNU PTH ein Thread mittels `yield` abgeben möchte, so wird zuerst in den Thread des Schedulers zurück gewechselt. Dort wird entschieden, welcher Thread nun an die Reihe kommen soll, um dann einen zweiten Kontextwechsel in diesen neuen Thread vorzunehmen (Stand Version 2.0.6).

Es ist allerdings auch möglich, die Schedulingalgorithmen in den Benutzerthreads auszuführen. Dann wird kein Thread für den Scheduler benötigt und der Wechsel zwischen zwei Threads findet direkt statt. Das folgende Programm ist der Nachweis, dass dies möglich ist. Es verwendet die GNU PTH Sub-API für die Kontextwechsel.

A.2.1 pth-fast.h

```
#ifndef __PTH_FAST_H
#define __PTH_FAST_H

#include <unistd.h>

#define MAX_NUMBER_OF_THREADS 10

// call this method first and once
void pthf_init();

// start scheduling. function returns if the last thread exists
void pthf_run();

// add a new thread for scheduling
void pthf_spawn(void(*func)(void*), void* arg, char* stack, size_t stacksize);

// let control be passed to another thread
void pthf_yield();

#endif
```

A.2.2 pth-fast.c

```
#include <stdlib.h>
#include <stdio.h>

#include <pth.h>
#include "pth-fast.h"

#define die pthf_abort(__LINE__)
```

```
// the round-robin queue
pth_uctx_t threads[MAX_NUMBER_OF_THREADS];

// index of the currently running thread
int current = -1;

// number of spawned threads
int number = 0;

// the context of the "main thread"
pth_uctx_t mainContext;

// the context of the exit handler
pth_uctx_t exitContext;

// local functions
void pthf_exit(void*);
void pthf_abort(int line);

void pthf_init()
{
    if (pth_uctx_create(&mainContext) == FALSE) die;

    if (pth_uctx_create(&exitContext) == FALSE) die;

    if (pth_uctx_make(exitContext, NULL, 16 * 1024,
          NULL, pthf_exit, NULL, NULL) == FALSE) die;
}

void pthf_run()
{
    if (number == 0) die;

    // pass control to the first thread
    current = 0;
    if (pth_uctx_switch(mainContext, threads[current]) == FALSE) die;

    // we reach here only, if the exit handler passes control to here.
    // This is only done, if no more threads are living.
    // So clean up the resources and return
    if (pth_uctx_destroy(exitContext) == FALSE) die;

    if (pth_uctx_destroy(mainContext) == FALSE) die;
}
```

```
void pthf_spawn(void(*func)(void*), void* arg, char* stack, size_t stacksize)
{
    if (number == MAX_NUMBER_OF_THREADS) die;

    // create and setup a new context
    pth_uctx_t context;
    if (pth_uctx_create(&context) == FALSE) die;
    if (pth_uctx_make(context, stack, stacksize,
            NULL, func, arg, exitContext) == FALSE) die;

    // enqueue the new thread
    threads[number++] = context;
}

void pthf_yield()
{
    // round robin
    int old = current;
    current++;
    if (current == number) current = 0;

    // switch directly to the next thread
    // this is where we save one context switch
    if (pth_uctx_switch(threads[old], threads[current]) == FALSE) die;
}

void pthf_exit(void* arg)
{
    int i;
    while(1) {
        // whenever we reach here a thread terminated and
        // control was therefore passed to here

        if (pth_uctx_destroy(threads[current]) == FALSE) die;
        number--;

        if (number == 0) {
            // the last thread did exit so terminate the whole scheduler
            // by passing control back to the pthf_run methode
            if (pth_uctx_switch(exitContext, mainContext) == FALSE) die;

            // not reached
            die;
        } else {
```

```
            // remove the thread from the queue
            for (i=current; i<number; i++) {
                threads[i] = threads[i+1];
            }
            if (current == number) {
                current = 0;
            }

            // pass control to the next thread
            if (pth_uctx_switch(exitContext, threads[current]) == FALSE) die;
        }
    }
}

void pthf_abort(int line)
{
    printf("aborting: %s:%d\n", __FILE__, line);
    abort();
}
```

A.2.3 Eine beispielhafte main.c

```
#include "pth-fast.h"

#include <stdio.h>
#include <unistd.h>

typedef struct {
    char* name;
    int count;
} Params;

void run(void* arg)
{
    int i;
    Params* p = (Params*) arg;
    for(i=0; i<p->count; i++) {
        printf("loop %d %s\n", i, p->name);
        sleep(1);
        pthf_yield();
    }
}
```

```
int main()
{
    Params p1 = {"t1", 2 };
    Params p2 = {"t2", 4 };

    pthf_init();

    pthf_spawn(run, &p1, NULL, 16 * 1024);
    pthf_spawn(run, &p2, NULL, 16 * 1024);

    pthf_run();

    return 0;
}
```

Anhang B

Dienste

Die Diplomarbeit von Stephanie Wist [41] behandelt das ICE Grid von Lethe (vlg. ICE Benutzerhandbuch, Kapitel 36 [21]). Zwei der Dienste in diesem Grid sind die Generatoren und der Breeze Server. In den folgenden Abschnitten findet sich eine kurze Anleitung zu diesen Diensten und ihrer Konfiuguration.

Da die Dienste Teil des Grids sind, werden sie automatisch gestartet und beendet. Außerdem verwenden sie fürs Logging von internen Fehlerereignissen nicht den Breeze Server. Statt dessen schreiben sie die Nachrichten auf ihre Standardausgaben, so dass diese als Ausgabe der Grid-*Nodes* erscheinen (vgl. ICE Benutzerhandbuch, Abschnitt 36.15.2 [21]).

B.1 Generator

Abschnitt 3.2.3 erklärt die Aufgaben des Generators und Abschnitt 3.2.5 führt aus, dass die Proxies Teil des Generatorprozesses sind. Zur Konfiguration des Breeze Servers existieren in der Deployment-Beschreibung des ICE Grids die folgenden Eigenschaften:

Config.WorkingDirectory Arbeitspfad, in dem die Simulationen übersetzt werden. Zu Beginn muss in diesem Verzeichnis der vollständige Quellcode des Frameworks enthalten sein.

Config.WaitpidTimeout Zeit in Sekunden, die der Proxy auf das Beenden eines Simulationsprozesses, der die Pipe zum Proxy geschlossen hat, maximal wartet. Nach Ablauf dieser Zeit wird der Simulationsprozess zurück gelassen und eine Fehlermeldung ausgegeben.

Config.ShutdownTimeout Zeit in Sekunden, die die Simulation vom Proxy maximal erhält, in Folge des `die` Kommandos zu beenden. Nach Ablauf dieser Zeit wird das Betriebssystem angewiesen, den Simulationsprozess zu beenden.

Config.KillTimeout Zeit in Sekunden, die das Betriebsssytem vom Proxy maximal erhält, den Simulationsprozess zu beenden. Nach Ablauf dieser Zeit wird der Simulationsprozess zurück gelassen und eine Fehlermeldung ausgegeben.

Config.MakeCmd Pfad zum GNU make Programm [25]

Darüberhinaus können in der Datei `global-config.mak`, die sich im Arbeitspfad befindet, die folgenden Einstellungen vorgenommen werden:

CC Pfad zum C++ Compiler

LD Pfad zum Linker

SLICEC Pfad zum Slicecompiler für C++ (vgl. ICE Benutzerhandbuch, Abschnitt 4.18 [21])

AR Pfad zum `ar` Programm zum Erstellen von statischen Bibliotheken

RANLIB Pfad zum `ranlib` Programm zum inidzieren von statischen Bibliotheken.

GLOBAL_CCFLAGS Optionen für den Compiler

GLOBAL_LDFLAGS Optionen für den Linker

GLOBAL_SLICEC_FLAGS Optionen für den Slicecompiler

B.2 Breeze Server

Abschnitt 3.2.6 erklärt die Aufgaben des Breeze Servers.

B.2.1 Nachrichten

Eine Nachricht besteht aus den folgenden Feldern:

Typ Der Typ der Nachricht. Mögliche Werte sind (vgl. Abschnitt 3.2.6)

- EMERG
- ALERT
- CRIT
- ERR
- WARNING
- NOTICE
- INFO
- DEBUG

Herkunft Ein beliebiger String, der die Herkunft einer Nachricht beschreibt.

Inhalt Ein beliebiger String, der den eigentlichen Inhalt der Nachricht wieder gibt.

Zeitstempel Die Unixzeit, zu dem die Nachricht beim Breeze Server einging. Das ist die Anzahl der vergangenen Sekunden seit dem 1. Januar 1970 00:00 h UTC.

B.2.2 Konfiguration

Zur Konfiguration des Breeze Servers existieren in der Deployment-Beschreibung des ICE Grids die folgenden Eigenschaften:

Breeze.Path Das Verzeichnis für die Logdateien.

Breeze.FileName Ein Präfix für den Namen der Logdateien.

Breeze.FileExtension Die Dateiendung der Logdateien.

B.2.3 Logdateien

Der Breeze Server legt für jeden Tag eine eigene Logdatei an. Der komplette Pfad einer Logdateien heißt:
`<Breeze.Path>/<Breeze.FileName>-JJJJ-MM-TT.<Breeze.FileExtension>`.
Alle Dateien, deren Pfad diesem Muster entsprechen, werden vom Breeze Server beim Starten automatisch als Logdateien interpretiert und verwendet. Während der Breeze Server läuft sollten diese Dateien nicht gelöscht werden.

B.2.4 publish

Mit dem Kommandozeilenprogramm "publish" kann der Benutzer Nachrichten an den Breeze Server senden. Die Kommandozeile sieht folgendermaßen aus:
`./publish.py (-p <proxy> | -n <locator>) -m <message> [-t <type>] [-o <origin>] [-h]`
Die Bedeutungen der Optionen sind

-p <proxy> Die Stringrepräsentation des ICE Proxies vom Breeze Servers. Diese Option kann nicht zusammen mit der Option -n verwendet werden.

-n <locator> Die Stringrepärsentation des ICE Proxies vom Namensdienst. Diese Option kann nicht zusammen mit der Option -p verwendet werden.

-m <message> Der Inhalt der Nachricht.

-t <type> Der Typ der Nachricht. Falls diese Option nicht gesetzt ist, hat die Nachricht den Typ DEBUG.

-o <origin> Die Herkunft der Nachricht. Falls diese Option nicht gesetzt wird, hat die Nachricht keine Herkunft.

-h Zeigt eine Hilfe zu den Optionen und beendet das Programm.

B.2.5 lastlog

Mit dem Kommandozeilenprogramm “lastlog” kann der Benutzer Lognachrichten vom Breeze Server erfragen. Dazu gibt er eine Reihe von Filterkriterien für die einzelnen Felder der Nachrichten an. Der Breeze Server liefert eine Nachricht genau dann aus, wenn sie alle Filter passiert.

Die vollständige Kommandozeile sieht folgendermaßen aus:

```
./lastlog.py (-p <proxy> | -n <locator>) [-i <type>[,<type>[,...]]]
[-e <type>[,<type>[,...]]] [-o <regex>] [-m <regex>] [-b <days> | -c
<timestamp>] [-s <days> | -d <timestamp>] [-f <seconds>] [-h]
```

Die Bedeutungen der Optionen sind:

-p <proxy> Die Stringrepräsentation des ICE Proxies vom Breeze Servers. Diese Option kann nicht zusammen mit der Option -n verwendet werden.

-n <locator> Die Stringrepärsentation des ICE Proxies vom Namensdienst. Diese Option kann nicht zusammen mit der Option -p verwendet werden.

-i <typliste> Nimmt eine Nachricht auf, wenn ihr Typ einem der Typen aus der Liste entspricht. Anstatt die vollständige Liste anzugeben, kann diese Option einfach weg gelassen werden.

-e <typliste> Nimmt eine Nachricht auf, wenn ihr Typ keinem der Typen aus der Liste entspricht. Wird die Option nicht gesetzt, wird kein Typ ausgeschlossen.

-o <regex> Nimmt eine Nachricht auf, wenn der reguläre Ausdruck auf die Herkunft der Nachricht passt. Wird die Option nicht gesetzt, wird auf das Herkunftsfeld nicht gefiltert.

-m <regex> Nimmt eine Nachricht auf, wenn der reguläre Ausdruck auf den Inhalt der Nachricht passt. Wird die Option nicht gesetzt, wird auf den Inhalt nicht gefiltert.

-b <days> Nimmt eine Nachricht auf, wenn sie innerhalb der letzten <days> Tage beim Breeze Server eingetroffen ist. Die Vorgabe ist ein Tag. Diese Option kann nicht zusammen mit der Option -c gesetzt werden.

-s <days> Nimmt eine Nachricht nicht auf, wenn sie innerhalb der letzten <days> Tage beim Breeze Server eingetroffen ist. Die Vorgabe sind 0 Tage. Diese Option kann nicht zusammen mit der Option -d gesetzt werden.

-c <timestamp> Nimmt eine Nachricht auf, wenn ihr Zeitstempel jünger ist als <timestamp>. Diese Option kann nicht zusammen mit der Option -b gesetzt werden.

-d <timestamp> Nimmt eine Nachricht nicht auf, wenn ihr Zeitstempel jünger ist als <timestamp>. Diese Option kann nicht zusammen mit der Option -s gesetzt werden.

-f <seconds> Diese Option aktiviert den follow-Modus. In diesem Modus wird der Breeze Server automatisch alle <seconds> Sekunden nach neuen Nachrichten befragt. Diese Option überschreibt die Optionen -s und -d.

-h Zeigt eine Hilfe zu den Optionen und beendet das Programm.

Anhang C

Verwendete Software

Zur Entwicklung von Lethe und zur Erstellung dieser Diplomarbeit wurde eine Reihe von freier Software eingesetzt, die ich an dieser Stelle nennen möchte.

Debian GNU/Linux `http://www.debian.org/`

Dia `http://www.gnome.org/projects/dia/`

GNU Compiler Collection `http://gcc.gnu.org/`

GNU make `http://www.gnu.org/software/make/`

gnuplot `http://www.gnuplot.info/`

LaTeX `http://www.latex-project.org/`

Subversion `http://subversion.tigris.org/`

The GNU Project Debugger `http://www.gnu.org/software/gdb/`

The Internet Communication Engine `http://zeroc.com/`

trac `http://www.edgewall.com/trac/`

Vim `http://www.vim.org/`

Vim-LaTeX `http://vim-latex.sourceforge.net/`

Xfig `http://www.xfig.org/`

Xpdf `http://www.foolabs.com/xpdf/`

Für die Übersetzung der LaTeX-Dateien wurde das TeX-Framework von Prof. Dr. Hermann Härtig der TU Dresden verwendet:
`http://os.inf.tu-dresden.de/~mp26/download/DA-latex-template.tgz`

Index

Ableitung, 11
Arbeitstyp, 37, 40
ARQ, 21
Assertion, 29
Ausnahme, 11

Basistyp, 37, 39
Beobachter, 14, 18
Blockgröße, 38
Breeze-Server, 33

explizites Beenden, 19

Feedback-Kante, 21

Generator, 31

Hypergraph, 15

ICE Objekt, 28
IceStorm, 33
impliziter Aufruf, 16
implizites Beenden, 19
initialer Knoten, 16

Kante, 15
Klasse, 10
Knoten, 15

lastlog, 34, 88

Modul, 35

Namensraum, 11

Operator überladen, 11

Phase, 14, 22
Port, 14, 22, 35
Protokollierer, 14, 19
Proxy, 32
publish, 34, 87
Puffertyp, 41

Runde, 14, 20

Scheduler, 16
Servant, 32
Simulationsbeschreibung, 13, 27
Simulationsobjekt, 29
stationärer Algorithmus, 18
Stream, 35, 36

Tausworthe, 51
Template, 11
Templates::Stream, 36

Vererbung, 11
Vorausschau, 45

Zwischenzustand, 25

Abbildungsverzeichnis

2.1 Abbildung eines gerichteten Hypergraphen auf einen gerichteten Graphen 10

3.1 Ein Simulationsgraph 15
3.2 Der Hypergraph des Fallbeispieles 15
3.3 Der Hypergraph eines ARQ Verfahrens 21
3.4 Initialisierung einer Feedback-Kante 22
3.5 Interaktion eines Knotens mit der Laufzeitumgebung 26
3.6 Zustandsautomat eines Simulationsobjektes 29

4.1 Der Hypergraph des Fallbeispieles mit Kennungen 55

5.1 Die Simulation für die Messungen 59
5.2 Der Verwaltungsaufwand in Abhängigkeit zur Blockgröße 64
5.3 Der Verwaltungsaufwand in Abhängigkeit zur Blockgröße - größere Skala 64
5.4 Vergleich mit und ohne Optimierung 65
5.5 Vergleich mit und ohne Optimierung - größere Skala 66
5.6 Erhöhung der Datenmenge - Vergleich mit und ohne Optimierung 67

Literaturverzeichnis

[1] *Beowulf*. http://www.beowulf.org/, April 2006.

[2] Bossert, M.: *Channel Coding for Telecommunications*. John Wiley & Sons, New York, 1. Auflage, 1999.

[3] Breitbach, M. und M. Bossert: *Digitale Netze*. Teubner Verlag, Stuttgart, 1. Auflage, 1999.

[4] Brian W. Kernighan, Dennis Ritchie: *Programmieren in C*. Hanser Fachbuch, 2. Auflage, Januar 1990.

[5] Brown, Willard A.: *SIMULA for those who know FORTRAN, PL/I or BASIC*. Norwegian Computing Center/SIMULA information publication, Norwegian Computing Center, Oslo, Norway, 1981.

[6] C. L. Lawson, R. J. Hanson, D. R. Kincaid und F. T. Krogh: *Basic Linear Algebra Subprograms for Fortran Usage*. ACM Transactions on Mathematical Software, 5(3):S. 308–323, September 1979.

[7] *CoCentric System Studio*. http://www.synopsys.com/, April 2006.

[8] *Debian GNU/Linux*. http://www.debian.org/, April 2006.

[9] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney und D. Sorensen, D: *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 3. Auflage, 1999.

[10] Engelschall, Ralf S.: *Portable Multithreading*. In: *USENIX Annual Technical Conference*, June 2000.

[11] Engelschall, Ralf S.: *GNU Pth – The GNU Portable Threads*. http://www.gnu.org/software/pth/, April 2006.

[12] *Free Software Foundation*. http://www.fsf.org/, April 2006.

[13] Fujimoto, Richard M.: *Parallel and Distributed Simulation Systems*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, INC., 2000.

[14] *GNU Compiler Collection*. http://gcc.gnu.org/, April 2006.

[15] *GoldSim*. http://www.goldsim.com/, April 2006.

[16] *GNU General Public Licence*. http://www.gnu.org/copyleft/gpl.html, April 2006.

[17] H. Partsch, W. Guttmann: *Grundlagen des Übersetzerbaus*. Vorlesungskript, University of Ulm, Oktober 2004.

[18] Herold, Helmut: *make. Das Profitool zur automatischen Generierung von Programmen*. Addison-Wesley, März 2003.

[19] High Performance Fortran Forum: *High Performance Fortran Language Specification*, 1992.

[20] Howatson, M.C.: *Reclams Lexikon der Antike*. Reclam, 2006.

[21] *The Internet Communication Engine*. http://www.zeroc.com/, April 2006.

[22] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington und R. Clint Whaley: *An updated set of Basic Linear Algebra Subprograms (BLAS)*. ACM Transactions on Mathematical Software, 28(2):S. 135–151, Juni 2002.

[23] Mark Galassi, Jim Davies, James Theiler Brian Gough Gerard Jungman Michael Booth Fabrice Rossi: *GNU Scientific Library: Reference Manual*. Network Theory Ltd, 2003.

[24] *Matlab*. http://www.mathworks.com/, April 2006.

[25] Mecklenburg, Robert: *GNU make*. O'Reilly, Mai 2005.

[26] *MLDesigner*. http://www.mldesigner.com/, April 2006.

[27] *Message Passing Interface*. http://www.mpi-forum.org/, April 2006.

[28] *GNU Octave*. http://www.octave.org/, April 2006.

[29] Payne, W. H.: *Fortran Tausworthe pseudorandom number generator*. Communications of the ACM, 13(1):S. 57–57, Januar 1970.

[30] *Ptolemy*. http://ptolemy.eecs.berkeley.edu/, April 2006.

[31] *Parallel Virtual Machine*. http://www.csm.ornl.gov/pvm/, April 2006.

[32] *Python*. http://www.python.org, April 2006.

[33] *Scilab*. http://www.scilab.org/, April 2006.

[34] *STLfit: An STL Error Message Decryptor for C++*. http://www.bdsoft.com/tools/stlfilt.html, April 2006.

[35] Stroustrup, Bjarne: *Die C++-Programmiersprache*. Addison-Wesley, 1. Auflage, 2000.

[36] Stroustrup, Bjarne: *Design and Evolution of C++*. Addison-Wesley, 2004.

[37] Tanenbaum, Andrew S.: *A Tutorial on Algol 68*. ACM Comput. Surv., Seiten 155–190, 1976.

[38] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. Carl Hanser Verlag München, 2. Auflage, 1995.

[39] Thomas Beuter, Peter Dadam: *Prinzipien der Replikationskontrolle in verteilten Systemen*. Ulmer Informatik-Berichte 95-11, University of Ulm, November 1995.

[40] Weinberg, Joshua: *Centralized Logging with* `syslog`. Sys Admin: The Journal for UNIX Systems Administrators, 7(10):S. 27, 30–34, Oktober 1998.

[41] Wist, Stephanie: *Entwurf und Implementierung eines verteilten Systems mit Benutzerschnittstelle zur Erstellung, Verteilung und Verwaltung von Simulationen in der Kanalcodierung*. Diplomarbeit, University of Ulm, Mai 2006.